

2

AD-A236 121



Carnegie Mellon University
Software Engineering Institute

Support Materials for

Formal Specification of Software

Support Materials SEI-SM-8-1.0

DTIC
ELECTE
JUN 03 1991
S C D

91-00921



Approved for public release;
Distribution Unlimited

91 5 31 002

Support Materials
for
Formal Specification of Software

SEI Support Materials SEI-SM-8-1.0

October 1987



Edited by

Alfs Berztiss
University of Pittsburgh

Acquisition For	
DTIC GRA21	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

Draft For Public Review

This technical report was prepared for the

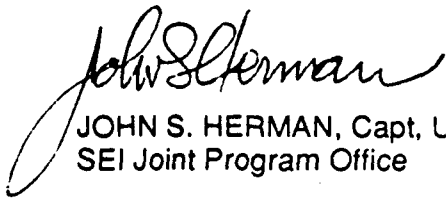
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1987 by Carnegie Mellon University.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Contents

Examples of Predicative Specifications	1
The SF (Set-Function) Methodology	11
SF Specification: Boat Hire	18
SF Specification: A Library System	26
SF Specification: An Elevator Controller	33
The Specification Process	43
Formal Specification Courses	61

Formal Specification of Software

Support Materials Revision History

Version 1.0 (October 1987) Draft for public review

Examples of Predicative Specifications

Alfs Berztiss
University of Pittsburgh

This section includes several annotated examples of specifications written in predicate calculus notation. Some of the details are left as exercises to the reader.

EXAMPLES OF PREDICATIVE SPECIFICATIONS

SMALLER EXAMPLES

The specification of a data transformer generally consists of two predicates. One describes the output independently of the input; the other relates the output to the input. For example, to specify sorting, we have to indicate both that the output is in fact sorted, and that the output is a permutation of the input:

Input: $X[1..n]$;

Output: $X'[1..n]$;

$Allop(\wedge; \{X'[i] \leq X'[i+1] \mid 1 \leq i < n\})$;

$permutation(X, X')$.

Notation: for any associative and commutative operator op , $Allop(op; set)$ indicates a distributive application of the operator over set . For example, if $S = \{5, 12, 14\}$, then $Allop(+; S) = 5+12+14 = 31$. The expression in \wedge given above is equivalent to $\forall i: 1 \leq i < n: X'[i] \leq X'[i+1]$. Often the set is defined by the use of a defining predicate: $\{s \mid P(s)\}$, read as "the set of all elements s such that the predicate P is true for s ." In our example the set consists of all statements $X'[i] \leq X'[i+1]$ such that $1 \leq i < n$. Note that $Allop(\vee; nullset) = false$ and that $Allop(\wedge; nullset) = true$.

Suppose that we wanted a function that returned the quotient q and remainder r of the division of the nonnegative integer x by the positive integer y . Here the two components of the output are related to the input by the expression

$$(x = y \times q + r) \wedge (0 \leq r < y).$$

Similarly we can specify a function that inverts a matrix A by relating its output B to A and to the identity matrix I : $AB = I$.

Another simple specification is that of the greatest common divisor. Here we define

$$\begin{aligned} \text{divides}(i : \text{Cardinal} ; x : \text{Natural}) &= \text{Allop}(\forall ; \{i \times q = x \mid 0 \leq q \leq x\}); \\ \text{gcd}(g : \text{Cardinal} ; x, y : \text{Natural}) &= \text{divides}(g, x) \wedge \text{divides}(g, y) \\ &\quad \wedge \text{not}(\text{Allop}(\forall, \{\text{divides}(i, x) \wedge \text{divides}(i, y) \mid g < i \leq \min(x, y)\})). \end{aligned}$$

A function can be defined in terms of *gcd*:

$$\text{fgcd}(x, y : \text{Natural}) = \text{getelement}(\{g \mid \text{gcd}(g, x, y)\}),$$

where *getelement* is a function that extracts an element from a set (here the set consists of a single element, namely the greatest common divisor of *x* and *y*).

Instead of defining *gcd* in terms of *divides*, we can specify it by means of three axioms:

$$\begin{aligned} \text{gcd}(x, x, 0); \\ \text{gcd}(g, x, y) &= \text{gcd}(g, x + y, y); \\ \text{gcd}(g, x, y) &= \text{gcd}(g, y, x). \end{aligned}$$

In mathematical language, we have here a *theory* of the greatest common divisor. From these axioms, together with the relation

$$x = y \times \text{div}(x, y) + \text{rem}(x, y),$$

can derive the predicative specification

$$y > 0 \rightarrow \text{gcd}(g, x, y) = \text{gcd}(g, y, \text{rem}(x, y)),$$

which can serve as a basis for an implementation of the *gcd* function.

A TEXT FORMATTER

The text formatter is one of the problems selected for study at the Fourth International Workshop on Software Specification and Design. Here the input is a string over the alphabet CH, and this string is to be split into lines. The input consists of words separated by sequences of *break characters*, namely sequences of blanks (BK) and linefeeds (LF). Let

$BC = \{BK, LF\}$. Then a word is a sequence of characters from $SC = CH - BC$ such that the character to the left of this sequence (if any) and the character to the right of the sequence (if any) belong to BC . The first word of the input may be preceded by characters from BC , and characters from BC may follow the last word of the input.

The output is to contain precisely the words of the input in precisely the order that they have in the input. The length of an output line is not to exceed the value *maxpos*. If the input contains a word that consists of more than *maxpos* characters, then the entire output is to be just the one single character BK . The first word of the output is not to be preceded by any characters from BC , and the last word is not to be followed by any such characters. The objective is to minimize the number of output lines. This objective is achieved if the output lines are built up in the order they have in the output, and for every line an attempt is made to pack as much of the remaining input into this line as the line can take. However, a specification is not to show bias toward a particular implementation. Therefore the description of the output will be totally declarative. This is particularly important here because the number of significantly different reasonable implementations is quite large.

We denote a string by $S(1)S(2)...S(N)$, where $length(S) = N$. Then let the input string be $B(1)B(2)...B(length(B))$, and the output string $C(1)C(2)...C(length(C))$. There are two parts to the specification. The first part consists of predicates that we consider of sufficiently general interest to be part of the data type of strings of words. The other part consists of predicates specific to this application.

Three of our predicates belong to the data type of strings of words: $word(S, i, j)$ is true if the character sequence $S(i)...S(j)$ defines a word; $word_number(S, k, i, j)$ is true if $S(i)...S(j)$ defines the k th word of S ; $word_count(S, k)$ is true if S contains k words.

$$\begin{aligned}
 word(S : String : i, j : 1..length(S)) = & \\
 & i \leq j \\
 & \wedge (i \neq 1) \rightarrow member(S(i-1), BC) \\
 & \wedge (j \neq length(S)) \rightarrow member(S(j+1), BC) \\
 & \wedge Allp(\wedge \{member(S(k), SC) \mid i \leq k \leq j\});
 \end{aligned}$$

$$\begin{aligned}
\text{word_number}(S: \text{String}; k, i, j: \text{Cardinal}) = & \\
& \text{word}(S, i, j) \\
& \wedge (k = 1) \rightarrow \text{Allop}(\wedge; \{ \text{member}(S(t), BC) \mid 1 \leq t < i \}) \\
& \wedge (k > 1) \rightarrow \text{Allop}(\vee; \{ \text{word_number}(S, k-1, u, v) \wedge \\
& \quad \text{Allop}(\wedge; \{ \text{member}(S(t), BC) \mid v < t < i \}) \\
& \quad \mid \text{member}(u, \text{Cardinal}) \wedge \text{member}(v, \text{Cardinal}) \});
\end{aligned}$$

$$\begin{aligned}
\text{word_count}(S: \text{String}; k: \text{Cardinal}) = & \\
& (k = 0) \rightarrow \text{Allop}(\wedge; \{ \text{member}(S(t), BC) \mid 1 \leq t \leq \text{length}(S) \}) \\
& \wedge (k > 0) \rightarrow \text{Allop}(\vee; \{ \text{word_number}(S, k, u, v) \wedge \\
& \quad \text{Allop}(\wedge; \{ \text{member}(S(t), BC) \mid v < t \leq \text{length}(S) \}) \\
& \quad \mid \text{member}(u, \text{Cardinal}) \wedge \text{member}(v, \text{Cardinal}) \});
\end{aligned}$$

Let us examine the predicate *word* in some detail. The four conjuncts in its definition establish, respectively, that limits *i* and *j* are properly related, that *S(i)...**S(j)* either starts at the left boundary of *S* or has a break character preceding it, that *S(i)...**S(j)* either ends at the right boundary of *S* or has a break character following it, and that no characters in *S(i)...**S(j)* are break characters. In the definition of *word_number* the first conjunct identifies the character sequence defined by *i* and *j* as a word. Then it is asserted that all characters preceding the first word are break characters. The sequence number of subsequent words is established recursively: it is asserted that there exist character positions *u* and *v* that define word *k-1*, and that all characters between the end of this word and word *k* are break characters. The interpretation of the definition of *word_count* is left as an exercise.

The next set of predicates relates to the application. Predicate *agrees* matches up the output words with the input words, and predicate *special_case* determines whether or not the input contains any word that is too long. Predicates *breaks_ok* and *lines_ok* relate to the output: *breaks_ok* establishes that there are no leading or trailing break characters, and that there is precisely one break character between each pair of words; *lines_ok* establishes the proper placement of linefeeds.

$agrees(B, C : String) =$

$Allop(\forall; \{word_count(B, k) \wedge word_count(C, k) \wedge$
 $Allop(\wedge; \{$
 $Allop(\forall; \{word_number(B, t, i, j) \wedge word_number(C, t, u, v) \wedge$
 $j - i = v - u \wedge$
 $Allop(\wedge; \{B(i + q) = C(u + q) \mid 0 \leq q \leq j - i\})$
 $\mid subset(\{i, j, u, v\}, Cardinal)\})$
 $\mid 1 \leq t \leq k\})$
 $\mid member(k, Cardinal)\})$;

$breaks_ok(C : String) =$

$not(member(C(1), BC))$
 $\wedge not(member(C(length(C)), BC))$
 $\wedge Allop(\wedge; \{member(C(j), BC) \rightarrow not(member(C(j+1), BC))$
 $\mid 1 \leq j < length(C)\})$;

$lines_ok(C : String) =$

$Allop(\wedge; \{C(i) \neq LF \mid 1 < i < length(C)\}) \rightarrow length(C) \leq maxpos$
 $\wedge Allop(\wedge; \{C(i) = LF \rightarrow$
 $(length(C) - i \leq maxpos \vee Allop(\forall; \{C(j) = LF \mid i < j \leq maxpos + i + 1\})) \wedge$
 $Allop(\forall; \{word(C, i + 1, k) \wedge i < maxpos \rightarrow k > maxpos \wedge i > maxpos \rightarrow$
 $Allop(\forall; \{C(q) = LF \wedge i - q - 1 \leq maxpos \wedge k - q > maxpos$
 $\mid 1 \leq q < i\})$
 $\mid i < k \leq length(C)\})$
 $\mid 1 < i < length(C)\})$;

$special_case(B : String) =$

$Allop(\forall; \{word(B, i, j) \mid (1 \leq i, j \leq length(B)) \wedge j - i \geq maxpos\})$;

$conversion_ok(B, C : String) =$

$special_case(B) \rightarrow (length(C) = 1 \wedge C(1) = BK)$
 $\wedge not(special_case(B)) \rightarrow (agrees(B, C) \wedge breaks_ok(C) \wedge lines_ok(C))$;

Predicate *agrees* asserts that strings *B* and *C* contain the same number of words, that corresponding words in *B* and *C* contain the same number of characters, and that corresponding characters are equal.

The definition of *lines^{ok}* is the most difficult to interpret. If the output string *C* contains no linefeeds, then it occupies just one line, and *length(C)* may not exceed *maxpos*. If *C* contains more than one line, then for every linefeed in *C* it has to be established that the end of the string or another linefeed is not too far away on the right. Further, if this is the first linefeed in *C*, then it has to be shown that the length of the substring between *C*(1) and the end of the first word that follows the linefeed exceeds *maxpos*; otherwise the length of this substring is measured from the preceding linefeed.

The specification of the text formatter was very difficult to write, and it is difficult to interpret. Moreover, one has to be careful to make the specification complete. For example, the assertion $j-i=v-u$ in predicate *agrees* is a later addition. Without it the strings "This man is my husband" and "This maniac is my husband" would be found to agree.

TWO-WAY MERGE

Consider input streams of records defined by the following key sequences.

A: 5 7 3 12 57 32 17 19 27 18 43 15

B: 2 9 11 8 15 30 42 20 35

Runs from the input streams are merged to produce output runs. Thus (5,7) and (2,9,11) yield (2,5,7,9,11); (3,12,57) and (8,15,30,42) yield (3,8,12,15,30,42,57); (32) and (20,35) yield (20,32,35). The merged runs go alternately into output streams *C* and *D*. At this point input stream *B* has been exhausted, but three runs still remain in stream *A*. The first and third go into *D*, the second into *C*. The output streams are

C: 2 5 7 9 11 20 32 35 18 43

D: 3 8 12 15 30 42 57 17 19 27 15

The two-way merge is defined in terms of the following predicates.

$$\begin{aligned}
run(A; i, j) &= Alloper(\wedge; \{ A[k] \leq A[k+1] \mid i \leq k < j \}) \\
&\wedge i > j \rightarrow A[i] < A[i-1] \\
&\wedge j < n \rightarrow A[j] < A[j+1];
\end{aligned}$$

$$\begin{aligned}
runnumber(A, k, i, j) &= run(A, i, j) \\
&\wedge k = 1 \rightarrow i = 1 \\
&\wedge k > 1 \rightarrow Alloper(\vee; \{ runnumber(A, k-1, u, i-1) \mid 1 \leq u < i \});
\end{aligned}$$

$$\begin{aligned}
runcount(A, k) &= (k = 0) \rightarrow null(A) \wedge \\
&(k > 0) \rightarrow Alloper(\wedge; \{ runnumber(A, k, u, n) \mid 1 \leq u \leq n \});
\end{aligned}$$

In our example above, there should be three runs on C . However, if we apply predicate *runcount* to C , we find that k is 2, i.e., the runs (2,5,7,9,11) and (20,32,35) have coalesced into a single run. This worsens matters to the extent that we need a complicated separate predicate to define output "runs".

$$\begin{aligned}
outrun(D, k, i, j) &= \\
¬(Alloper(\vee; \{ runnumber(A, k \times 2, u, v) \mid v \leq size(A) \wedge u \leq v \})) \rightarrow \\
&\quad Alloper(\vee; \{ runnumber(B, k \times 2, s, t) \wedge D[i..j] = B[s..t] \mid t \leq size(B) \wedge s \leq t \}) \\
&\wedge not(Alloper(\vee; \{ runnumber(B, k \times 2, u, v) \mid v \leq size(B) \wedge u \leq v \})) \rightarrow \\
&\quad Alloper(\vee; \{ runnumber(A, k \times 2, s, t) \wedge D[i..j] = A[s..t] \mid t \leq size(A) \wedge s \leq t \}) \\
&\wedge Alloper(\vee; \{ runnumber(A, k \times 2, u, v) \wedge runnumber(B, k \times 2, s, t) \\
&\quad \mid v \leq size(A) \wedge u \leq v \wedge t \leq size(B) \wedge s \leq t \}) \rightarrow \\
&\quad Alloper(\vee; \{ merge(D[i..j], A[u..v], B[s..t]) \\
&\quad \mid v \leq size(A) \wedge u \leq v \wedge t \leq size(B) \wedge s \leq t \});
\end{aligned}$$

$$outok(D, k) = Alloper(\vee; \{ outrun(D, k, i, j) \mid 1 \leq i, j \leq size(D) \});$$

Analogous definitions can be written for C . The definition of predicate *merge* is left as an exercise. The two-way merge itself may now be specified.

$$\begin{aligned}
two_way_merge(A, B, C, D) &= \\
&Alloper(\wedge; \{ outok(C, i) \mid 1 \leq i \leq div(max(frunccount(A), frunccount(B))+1, 2) \}) \\
&\wedge Alloper(\wedge; \{ outok(D, i) \mid 1 \leq i \leq div(max(frunccount(A), frunccount(B)), 2) \});
\end{aligned}$$

where

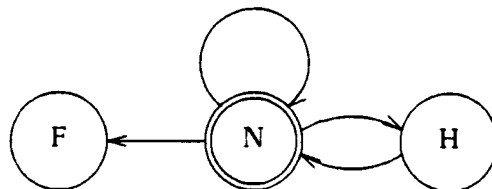
$$f_{runcount}(A) = getelement(\{k \mid runcount(A, k)\});$$

Sorting by the two-way merge is accomplished by applying the operation specified by predicate *two-way-merge* first to *A* and *B*, then to *C* and *D*, then to *A* and *B* again, and so forth until one of the output streams is empty. However, the predicative specification is very complicated, and it may be worthwhile to try a different approach.

We propose a generator that delivers the next value of a data stream each time it is invoked, and further propose that tags be associated with the values delivered by the generator. There are to be three tag values: N(ormal), H(old), and F(inished). The tag value is H after the generator has delivered the last item of a run, unless this is the last item of the entire input stream, in which case the value is F. Otherwise the value is N. Let us now assume that the generator has state. Then the behavior of the generator can be described in terms of state transitions.

- N Stay in state N, or change to H or F. If new state is N, advance to next item in input stream.
- H Change state to N, advance to next item in input stream.
- F Stay in state F.

The permissible state transitions are shown in the diagram below. The node representing state N is doubly circled because the generator starts in this state (unless the input stream is empty, in which case it starts in state F).



The tag values make it easy to specify a program that accepts data from the two input streams and moves these data into the two output streams. We assume that the input from the generator over *A* is in location *Aval*, and the input from the generator over *B* in *Bval*. Each record is assumed to contain a field *key*.

Tag-action table for a two-way merge

AB Action

NN If *Aval* < *Bval* then output *Aval* and call generator over *A*, else output *Bval* and call generator over *B*

NH Output *Aval* and call generator over *A*

NF Output *Aval* and call generator over *A*

HN Output *Bval* and call generator over *B*

HH Call generator over *A* and *B*; switch output streams

HF Call generator over *A*; switch output streams

FN Output *Bval* and call generator over *B*

FH Call generator over *B*; switch output streams

FF HALT

The main point is that the definition of the generator and the tag-action table constitute a full specification of the two-way merge. The specification is abstract in the sense that it is independent of the nature of the input streams (files, linear arrays, instances of some other structure). Moreover, the specification easily generalizes to a three-way or a four-way merge. Only the table that specifies the merge needs to be changed (to one with 27 or 81 entries, respectively). This table translates immediately into a program composed of nested case statements.

The SF (Set-Function) Methodology

Alfs Berztiss
University of Pittsburgh

An overview of the set-function methodology is given. This should be read before the examples that follow in this collection. The discussion includes advice about the construction of specifications as well as a description of the properties of the resulting products.

THE SF (SET-FUNCTION) METHODOLOGY

A specification methodology for information-control systems should have a sound theoretical base, and it should be consistent with the approaches used in the specification of other software elements. The SF methodology has both these properties. It is based on sets and functions, which are well understood mathematical concepts. Further, since the essence of data abstraction in general is that data types are defined in terms of sets and functions, SF is consistent with the general principles of data abstraction. Moreover, because an SF data type and the events associated with it define each other, the specification is self-contained. The SF methodology has been used to write the specifications of quite a number of software systems, among others the IFIP Working Conference example, which can be regarded as a standard benchmark, and a system for handling bank accounts. Note, however, that the IFIP Working Conference example was written while SF was still being developed, and has the serious flaw of not being modularized.

An SF specification consists of one or more segments. Each segment has three components:

- a schema definition,
- specifications of events,
- a responder that consists of transactions.

The schema definition identifies a set as being the primary set of interest for the segment, e.g., a set of bank accounts or a set of library books or a set of persons. In addition to the primary set there may be secondary sets, which for the most part merely provide a range for a function. For example, in a case study discussed below, library books are tagged with indicators of their subject matter. These indicators are defined as a secondary set. The schema definition also defines a set of functions (finite maps), which, for the most

part, have the primary set of interest for their domain. Examples of functions for bank accounts: balance in an account, overdraft limit, transactions for the current month (a set-valued function). Some functions have a null domain—they represent properties that pertain to the entire segment. Constant functions for the bank accounts: interest rate, number of transactions for which the bank does not charge, the charge for each excess transaction.

Some functions are *defined functions*. They can be defined in terms of other functions. For example, we could have a predicate *referees* such that *referees*(*r*, *p*) is true if referee *r* has been assigned as a referee to paper *p*. Then the function *refs of paper*, which is to map from the set of papers to the power set of the set of referees, i.e., which is to return the set of the referees of a paper, can be defined in terms of the predicate *referees*:

$$\text{refs of paper}(x) = \{y : \text{referees}(y, x)\};$$

Functions that are not defined are called *basic functions*.

The specification of events consists of preconditions and postconditions. Preconditions determine under what circumstances an event may take place. They serve as a check on the feasibility of input values, and embody consistency criteria for the data base of the information system. Postconditions are subdivided into setconditions, mapconditions, and sigconditions. Setconditions and mapconditions indicate the changes that sets and maps undergo in consequence of an event taking place. Sigconditions send signals to the responder in the form of raised flags.

Our example of an event assigns a referee to a paper. The preconditions test that the argument *p* does indeed belong to the set of submitted papers *S*, and that *ref* is an active referee. The prefix *R* indicates that *active* has been defined in the segment whose primary data set is *R* (i.e., the set of referees); similarly for prefixes *CO* (conference organization) and *P* (persons). The other preconditions test that the number of papers assigned to *ref* does not exceed a limit, that *ref* does not belong to the same organization as any of the authors of *p*, and that the number of referees for the paper does not exceed a limit. The mapcondition adjusts predicate *referees*. Primed entities refer to values after the event.

such as the primed *referees* here. Note very carefully that postconditions are assertions rather than assignments. The operational interpretation of a postcondition: the *minimal* modification of the data base that makes the assertion hold. The sigconditions set flags: flag *paper*~*to*~*ref* is to initiate the actual sending out of the paper to the referee; flag *find*~*refs* remains on as long as the limit on the number of referees has not been reached for this paper.

EVENT *Assign*~*referee*~*to*~*paper*(*p*, *ref*);

PRECONDITIONS— *member* (*p*, *S*);
 R. active (*ref*);
 ref ~*paper* ~*count* (*ref*) < *CO. ref* ~*paper* ~*limit* ;
 $\text{Allop}(\wedge; \{ \text{not}(\text{P. affiliation}(\text{ref}) = \text{P. affiliation}(x)) \mid$
 member (*x*, *authors* (*p*))});
 card (*refs* ~*of* ~*paper* (*p*)) < *CO. ref* ~*number* ;
 MAPCONDITIONS— *referees*'(*p*, *ref*) = true;
 SIGCONDITIONS— (*paper* ~*to*~*ref* (*p*, *ref*))ON;
 card (*refs* ~*of* ~*paper* '(*p*))= *CO. ref* ~*number*
 → (*find* ~*refs* (*p*))OFF;

ENDEVENT;

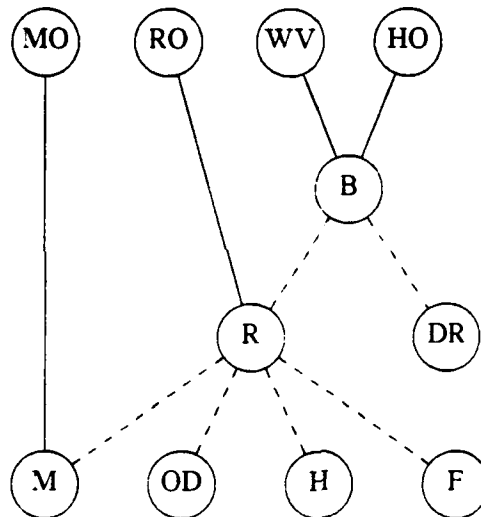
Let us now look again at defined functions. Defined functions have two purposes. First, they may be required in the specification itself, as is *refs*~*of*~*paper* in the specification of our event *Assign*~*referee*~*to*~*paper*. Second, every query can be regarded as no more than a request for the evaluation of a function. We can anticipate some queries, and the answer to an anticipated query can be predefined: if it is the value of a basic function, nothing needs to be done; otherwise a defined function that will provide an answer to the query is introduced in the schema definition.

In the SF information base all persistent data objects belong to sets and maps. The sets are arranged in hierarchies, and SF provides multiple inheritance. Multiple inheritance can be of two kinds. First, an object of type *X* can inherit properties of both a type *Y* and another type *Z*. Second, some objects of type *X* may inherit the properties of type *Y*, while

other objects of type X may inherit properties of type Z . In SF the first kind of inheritance is handled by multiple application of the ISA, and the second by set partitioning and ISA. Thus, for a boat hiring example to be discussed in detail further on, the dependencies defined by

TYPE B (SUBSETS: R (SUBSETS: F, H, OD, M ISA *Maintenance~object*)
 ISA *Registered~object*, DR)
 ISA *Water~vehicle* ISA *Hire~object*;

have the graphical representation:



In this diagram ISA links are represented by solid lines; subset relationships by dashed lines. The subset names R and DR are abbreviations for *Registered* and *Deregistered*, respectively. Subset names M , OD , H , and F refer to boats that are in maintenance, overdue (i.e., were not returned when expected), hired, and free for hire.

The usefulness of inheritance depends on how easy it is to create a new type from its parent types. Suppose that the parent types have available functions f_1, f_2, \dots, f_k , and the declaration of a new type T contains functions $f_1, \dots, f_k, f_{k+1}, \dots, f_n$. Then type T inherits functions f_1, f_2, \dots, f_k . While functions f_1, \dots, f_{k-1} may not be changed by

events in the segment of T , functions f_1, \dots, f_k may be changed, but the changes are not propagated back to the parent segments.

The inability to make changes in a parent type is a special case of the general principle that no SF event may directly change an object that does not belong to the segment in which the event resides. Suppose that type B is the basis type for a segment *Boat^{hire}*. Then no event in this segment may change anything in any other segment, say one that deals with repair of boats. However, we may import types, and declare instances of the imported type locally in the importing type. Changes may then be brought about in the locally declared instances. For example, in an airline reservation system customers may have to be put into waiting lines. The type of queue would be imported. It would provide the operations, but the actual queues of customers would reside in the reservation system as locally declared instances.

But changes can be brought about in other segments indirectly. This is by message passing. An SF message (or flag) is a signal, which may be provided with arguments. A signal, after it has been raised by an event, is picked up by a responder transaction, either in the same segment, or, in case the signal is declared as exported, in another segment. A signal remains alive until it is explicitly turned off. A variant of signals consists of counters, which may be incremented or decremented. A counter remains alive until its value drops to zero. Signals also provide the means of communication between segments. Thus, segment A may request segment B to perform some action by raising a flag. This flag is *exported* by segment A and *imported* by segment B. Segment B may inform segment A of the completion of the action by an explicit second signal, or merely by turning off the flag. The interleaving of events and transactions make SF processes equivalent to Petri nets, with the manipulation of signals and counters corresponding to the movement of tokens in a Petri net.

The responder processes signals immediately, at some specific time, e.g., 06:30 every morning, or at set intervals. The responder initiates events on its own accord, prompts the

user to initiate events, or reminds the user that some action is to be performed. For example, in a system that manages bank accounts, a withdrawal event may cause the account to become overdrawn. In such a case an overdraft signal is to be sent to the responder. A transaction in the responder now initiates an event that assesses a penalty charge against the account, and reminds the account manager to send out an overdraft notice to the customer. No event may initiate another event directly--all such initiations have to be carried out via transactions. Events initiated by the responder are called internal events; all other events are external. Internal events have no preconditions--all input data associated with such events are assumed to have been already checked. Sometimes, however, the responder establishes that external events are to be initiated. In such a case the responder issues prompts to the user. For example, in the assigning of referees to a paper, the responder prompts the user to initiate referee assignment events until the required number has been reached. Reminders issued by transactions do not in general relate to events.

The separation of the action of sending out a message (i.e., raising a signal as part of an event) from the definition of the ultimate effect of this message (in a responder, which may reside in a different segment) has several pleasing features. First, the events can be defined independently of the responder, and all decisions regarding the precise effect of a message can be postponed. Second, because messages are not addressed, more than one transaction can pick up a message, which adds to the flexibility of the entire system. Third, all indications of the time dependence of the effect of a message belong to the definition of the responder, i.e., the specification of time-related aspects of the system is confined to responders.

SF Specification: Boat Hire

Alfs Berztiss
University of Pittsburgh

A (relatively) simple problem is posed and solved in set-function notation.

SF SPECIFICATION: BOAT HIRE

Our first example of an SF specification relates to an enterprise that hires out boats. Hiring hours are 9:00 to 20:00, and all boats must be back by 21:00. The set of interest is that of boats (B), which is partitioned into the subsets of registered (R) and deregistered (DR) boats. The latter are no longer in use, i.e., have been scrapped. The set of registered boats is partitioned into free (F), hired (H), overdue (OD), and "maintained" (M) boats. The boats in set M are to be inspected, and the inspection indicates that they are free of defects, to be repaired, or beyond repair. The labels in secondary set S^{code} correspond to these possibilities. The majority of the events move boats from one of these subsets into another. They are *Hire* (which moves a boat from subset F to subset H); *Return* (from H to F or to M); *Mark~o~due* (from H to OD , applied to all boats that have not been returned at some specified time after 21:00); *Recover* (from OD to M); *Inspect* (from F to M); *Maintenance~check* (from M to F); *Reinstate* (from M to F).

Event *Buy~boat* brings a boat into the enterprise; event *Deregister~boat* takes it out. The purpose of *Needs~analysis* is to update a parameter that indicates how many additional boats the enterprise needs. Events *Deregister~boat*, *Mark~o~due*, and *Reinstate* differ from the others in that they are initiated by the system itself, i.e., they are *internal*. Events may have certain control functions. For example, when a boat is returned, and it had been hired out for 200 hours or more since its last inspection, the sigcondition *Check~condition* asks the responder to initiate a maintenance inspection of the boat. Such an inspection is mandatory when an overdue boat has been recovered.

We would like to answer rather complicated queries, such as "What is the total time for which boat x has been hired this month?" The functions of the type B serve mostly this purpose. For example, *Latest~hire* indicates the time at which the latest hiring event took place for the given boat, and H^{set} returns a set of triples indicating for each regular

hiring event the date and time the event took place, as well as the length of the interval for which the boat had been out.

These functions require the importation into the segment under consideration of several predefined types: T^{min} , with function $T^{now^{min}}$ that returns the current time with a resolution of one minute, and in which comparisons are standard operations; $Date$, again with comparisons, and the function D^{now} that returns the current date; $T^{dur^{min}}$, which consists of durations of time intervals measured in minutes, and in which subtraction of times, dates, or of pairs consisting of dates and times are standard operations.

Notation: In the specification of an event a primed quantity indicates a value after the event, an unprimed quantity indicates a value before the event. X -set stands for the power set of X . Let us also interpret the notation in the specification of the functionality of H^{sigma} . This is a defined function in the sense that it can be constructed from some other function, namely H^{set} here. The function $Coord$ belongs to the basic type of maps--here it extracts the third coordinate from each triple in the range of H^{set} .

SEGMENT $Boat^{hire}$;

IMPORTED TYPE T^{min} ENDTYPE;

IMPORTED TYPE $Date$ ENDTYPE;

IMPORTED TYPE $T^{dur^{min}}$ ENDTYPE;

TYPE B (SUBSETS: R (SUBSETS: F, H, OD, M ISA $Maintenance^{object}$), DR)
ISA $Water^{vehicle}$ ISA $Hire^{object}$;

SECONDARYSETS- $S^{code} = \{"ok", "repair", "scrap"\}$;

FUNCTIONS- $Boat^{in}$: $B \rightarrow Date$;

$Latest^{hire}$: $R \rightarrow T^{min}$;

H^{set} : $R \rightarrow (Date \times T^{min} \times T^{dur^{min}})\text{-set}$;

H^{sigma} : $R \rightarrow T^{dur^{min}}$: $H^{sigma}(b) =$
 $Allop(+, Coord(H^{set}(b), 3))$;

H^{hist} : $B \rightarrow (Date \times T^{min} \times T^{dur^{min}})\text{-set}$;

OD^{set} : $B \rightarrow (Date \times T^{min})\text{-set}$;

$REC_set : B \rightarrow (Date \times T_min) \text{--}set;$
 $INSP_set : B \rightarrow Date \text{--}set;$
 $Inspection_code : M \rightarrow S_code;$
 $Boat_shortage : \rightarrow Integer;$

ENDTYPE;

EVENT *Buy*(*boat*):

PRECONDITIONS— $not(Member(boat, B));$

SETCONDITIONS— $F' = F \cup \{boat\};$

MAPCONDITIONS— $Boat_in'(boat) = D_now;$
 $Boat_shortage' = Boat_shortage - 1;$

ENDEVENT;

INTERNAL EVENT *Deregister*(*boat*):

SETCONDITIONS— $R' = R - \{boat\};$

$DR' = DR \cup \{boat\};$

MAPCONDITIONS— $Boat_shortage' = Boat_shortage + 1;$

ENDEVENT;

EVENT *Needs*(*k*: Integer):

MAPCONDITIONS— $Boat_shortage' = Boat_shortage + k;$

ENDEVENT;

EVENT *Hire*(*boat*):

PRECONDITIONS— $Member(boat, F);$

$T_now_min \leq 20:00;$

SETCONDITIONS— $F' = F - \{boat\};$

$H' = H \cup \{boat\};$

MAPCONDITIONS— $Latest_hire'(boat) = T_now_min;$

ENDEVENT;

EVENT *Return*(*boat*):

DEFINITIONS— $t_dur : T_now_min - Latest_hire(boat);$

PRECONDITIONS— $Member(boat, H);$

SETCONDITIONS— $H' = H - \{boat\};$

$H \sim \sigma'(boat) \geq 200:00 \rightarrow M' = M \cup \{boat\};$
 $H \sim \sigma'(boat) < 200:00 \rightarrow F' = F \cup \{boat\};$
 MAPCONDITIONS- $H \sim \sigma'(boat) = H \sim \sigma(boat) \cup \{<D \sim now, Latest \sim hire(boat), t \sim dur>\};$
 SIGCONDITIONS- $H \sim \sigma'(boat) \geq 200:00 \rightarrow (Check \sim condition(boat))ON;$
 ENDEVENT;

INTERNAL EVENT *Mark \sim o \sim due(boat);*

SETCONDITIONS- $H' = H - \{boat\};$
 $OD' = OD \cup \{boat\};$
 MAPCONDITIONS- $OD \sim \sigma'(boat) = OD \sim \sigma(boat) \cup \{<D \sim now, Latest \sim hire(boat)>\};$
 ENDEVENT;

EVENT *Recover(boat);*

PRECONDITIONS- *Member(boat, OD);*
 SETCONDITIONS- $OD' = OD - \{boat\};$
 $M' = M \cup \{boat\};$
 MAPCONDITIONS- $REC \sim \sigma'(boat) = REC \sim \sigma(boat) \cup \{<D \sim now, T \sim now \sim min>\};$
 SIGCONDITIONS- $(Check \sim condition(boat))ON;$
 ENDEVENT;

EVENT *Inspect(boat);*

PRECONDITIONS- *Member(boat, F);*
 SETCONDITIONS- $F' = F - \{boat\};$
 $M' = M \cup \{boat\};$
 SIGCONDITIONS- $(Check \sim condition(boat))ON;$
 ENDEVENT;

EVENT *Maintenance \sim check(boat, status: S \sim code);*

SETCONDITIONS- *status = "ok" \rightarrow*
 BLOCK $M' = M - \{boat\};$
 $F' = F \cup \{boat\};$
 ENDBLOCK;
 MAPCONDITIONS- $H \sim hist'(boat) = H \sim hist(boat) \cup H \sim \sigma(boat);$
 $H \sim \sigma'(boat) = Nullset;$
 $INSP \sim \sigma'(boat) = INSP \sim \sigma(boat) \cup \{D \sim now\};$

SIGCONDITIONS— $status = "repair" \rightarrow (Repair \sim boat(boat))ON;$
 $status = "scrap" \rightarrow (Scrap \sim boat(boat))ON;$
 $(Check \sim condition(boat))OFF;$

ENDEVENT;

INTERNAL EVENT $Reinstate(boat);$

SETCONDITIONS— $M' = M - \{boat\};$
 $F' = F \cup \{boat\};$

ENDEVENT;

TRANSACTION $Maintenance;$

$@(T \sim min \sim now): Forall(x): Member(x, M): ON(Check \sim condition(x))ON-$
 $PROMPT(Maintenance \sim check : x);$

ENDTRANSACTION;

TRANSACTION $O \sim due \sim check;$

$@(21:15): Forall(x): Member(x, H): Mark \sim o \sim due(x);$

ENDTRANSACTION;

TRANSACTION $Check \sim needs;$

$Member(D \sim now + 1, EOM \sim dates) \rightarrow PROMPT(Needs \sim analysis);$

ENDTRANSACTION;

TRANSACTION $Boat \sim purchase;$

$Member(D \sim now, EOM \sim dates) \rightarrow PROMPT("Buy boat"); TIMES(Boat \sim shortage);$

ENDTRANSACTION;

TRANSACTION $Boat \sim repair;$

(* This transaction is triggered by $Repair \sim boat$; it signals some other segment-- the actual repairs are undertaken in this other segment, and depend on availability of resources; the flag $Boat \sim repaired$ of transaction $Back \sim in \sim service$ is set by some event in the other segment. *)

ENDTRANSACTION;

TRANSACTION *Back in service*:

@(T_{min}^{now}): Forall (x): Member (x, M): ON(*Boat repaired* (x))OFF— *Reinstate* (x);

ENDTRANSACTION;

TRANSACTION *Send out searchers*:

Forall (x): Member (x, OD): REMIND("Find boat": x);

ENDTRANSACTION;

TRANSACTION *Scrapping of boat*:

Forall (x): Member (x, M): ON(*Scrap boat* (x))OFF— *Deregister boat* (x);

ENDTRANSACTION;

TRANSACTION *Deregistration test*:

Forall (x): Member (x, OD): $D^{now} -$

$Allop(max, \{y \mid y = Coord(OD^{set}(x), 1)\}) > 7 \rightarrow Deregister\ boat(x)$;

ENDTRANSACTION;

ENDSEGMENT;

Transactions are of two types. First, transactions that are to take place at a given time are marked with the symbol @, e.g., @(21:15), @(T_{now}^{min}). In our example one marked transaction is to take place at 21:15, at which time the event *Mark overdue* is to be initiated for all hired boats that are still out. Unmarked transactions are performed according to a fixed schedule, which is once a day in our case. E.g., for each overdue boat, a reminder is issued that this boat is to be looked for. As another part of this daily procedure the current date is compared against end-of-month dates (in set *EOM dates*, which belongs to type *Date*), and transactions *Boat purchase* and *Check needs* are initiated on an end-of-month date or the day preceding it, respectively. Typically, a transaction initiates an event, issues a reminder, or issues a prompt. Reminders do not in general relate to events. On the other hand, prompts ask the user to initiate events. These events are not external, but the

system cannot initiate them on its own. For example, the purpose of event *Maintenance~check* is to determine what is to be done to a boat on the basis of the value of *status*. However, the user has to supply this value.

SF Specification: A Library System

Alfs Berztiss
University of Pittsburgh

One of the problems from the Fourth International Workshop on Software Specification and Design is described and solved in set-function notation. (Problems from this workshop appear often in the literature—instructors are encouraged to look for comparative examples in other notations.)

SF SPECIFICATION: A LIBRARY SYSTEM

The library system is another of the problems selected for study at the Fourth International Workshop on Software Specification and design. A basic requirement is that the library is to provide for multiple copies of particular titles. This means that a distinction has to be maintained throughout between *titles* of books and *copies* of books. We shall use the term *book* only when this term could indicate either a title or a copy. An informal statement of additional requirements and constraints on the system:

- copies are added to the library and are removed;
- copies are checked out and returned by borrowers;
- every copy is either in the library or else it is checked out to a borrower;
- no more than a predefined number of copies may be checked out to a borrower.

The system is to have these minimal query-answering capabilities:

- listing of books by a particular author or in a particular subject area;
- listing of copies that are checked out to a given borrower (restricted to library staff except that borrowers may find out what copies they themselves have borrowed);
- indication of the name of the borrower who last checked out a particular copy (restricted to library staff).

The distinction between titles and copies indicates the need for at least two segments in the system. The segment for titles then relates to information that is common to all copies of a particular book, such as the author or authors and the subject areas of the book. Addition or removal of copies can have an impact on the titles segment. Thus, when the very first copy of a particular title arrives at the library, appropriate information has to be added in this segment. Again, when the last copy of a title is removed, the information

regarding this title is not deleted, but is archived. The archiving is implemented by the partitioning of the set of titles into subsets *INCAT* and *HASBEEN*, and moving the title from *INCAT* into *HASBEEN*. Should a new copy of the book become again part of the library holdings, then the title is restored to *INCAT*.

The primary event is the addition of a copy to the library, but this event requires that the title be already registered in the titles segment. Consequently the addition of a first copy is somewhat circuitous: the copies segment signals the titles segment that a new title is to be added; after the title has been added, a signal goes back to the copies segment to initiate addition of the copy; the copy is finally added to the catalog.

SEGMENT *Titles*:

IMPORTED SIGNALS *Add~title, Drop~title, Move~title*;

EXPORTED SIGNALS *Catalog~copy*;

IMPORTED TYPE *Author* ENDTYPE;

(* Some types, such as *Integer*, *Boolean*, and *Text* are assumed to be universally available. The form *Subject~area: Area* used below indicates that *Area* is an abbreviation of *Subject~area*. *)

TYPE *Title*: *T* (SUBSETS: *INCAT, HASBEEN*);

SECONDARY SETS- *Subject~area: Area*;

FUNCTIONS— *title~text* : *T* → *Text* ;
 authors : *T* → *Author* -set;
 subjects : *T* → *Area* -set;

ENDTYPE;

EVENT *Add~title(newcopy; book; t: Text; A: Author-set; S: Area-set)*;

(* The types of arguments *t*, *A* and *S* are known, and are indicated in the list of arguments; *book* has not yet been added to set *T*, i.e., it has no type; *newcopy* is an argument that is being passed through from the copies segment back to the copies segment, where it will become part of the set *C*. *)

PRECONDITIONS— not(*member (book, T)*);

SETCONDITIONS— *INCAT'* = *INCAT* ∪ {*book*};

MAPCONDITIONS— $title \sim text'(book) = t;$

$authors'(book) = A;$

$subjects'(book) = S;$

SIGCONDITIONS— $(Catalog \sim copy(newcopy, book))ON;$

ENDEVENT;

INTERNAL EVENT *Reactivate(newcopy, book);*

SETCONDITIONS— $INCAT' = INCAT \cup \{book\};$

$HASBEEN' = HASBEEN - \{book\};$

SIGCONDITIONS— $(Catalog \sim copy(newcopy, book))ON;$

ENDEVENT;

INTERNAL EVENT *Drop \sim title(book);*

SETCONDITIONS— $INCAT' = INCAT - \{book\};$

$HASBEEN' = HASBEEN \cup \{book\};$

ENDEVENT;

TRANSACTION;

@ ($T \sim min . now$): ON(*Add \sim title(newcopy, book)*)OFF:

PROMPT(*Add \sim title : newcopy, book*);

ENDTRANSACTION;

TRANSACTION;

@ ($T \sim min . now$): ON(*Drop \sim title(book)*)OFF: *Drop \sim title(book)*;

ENDTRANSACTION;

TRANSACTION;

@ ($T \sim min . now$): ON(*Move \sim title(newcopy, book)*)OFF: *Reactivate(newcopy, book)*;

ENDTRANSACTION;

ENDSEGMENT;

SEGMENT *Copies*:

IMPORTED SIGNALS *Catalog~copy*: (* This signal may also be set locally. *)

EXPORTED SIGNAL *Drop~title*, *Move~title*, *Add~title*;

IMPORTED TYPE *Title*: *T* ENDTYPE;

IMPORTED TYPE *Borrower*: *B* ENDTYPE;

TYPE *Copy*: *C*;

(* Values in parentheses are starting values. *)

FUNCTIONS— *book~id*: $C \rightarrow T$;
 borrowed: $C \rightarrow \text{Boolean}(\text{false})$;
 last~out: $C \rightarrow B(\text{nil})$;
 books~out: $B \rightarrow \text{Integer}(0)$;
 limit: $\rightarrow \text{Integer}(0)$;

ENDTYPE;

EVENT *Set~limit*(*k*: *Integer*);

MAPCONDITIONS— *limit*' = *k* ;

ENDEVENT;

EVENT *Check~copy*(*newcopy*, *book*);

SIGCONDITIONS— *member* (*book* , *INCAT*) \rightarrow (*Catalog~copy* (*newcopy* , *book*))ON;
 member (*book* , *HASBEEN*) \rightarrow (*Move~title* (*newcopy* , *book*))ON;
 not(*member* (*book* , *T*)) \rightarrow (*Add~title* (*newcopy* , *book*))ON;

ENDEVENT;

INTERNAL EVENT *Add~copy*(*newcopy*, *book*);

SETCONDITIONS— $C' = C \cup \{\text{newcopy}\}$;

MAPCONDITIONS— *book~id*'(*newcopy*) = *book* ;

ENDEVENT;

EVENT *Remove~copy*(*copy*);

PRECONDITIONS— *member* (*copy* , *C*);
 not(*borrowed* (*copy*));

SETCONDITIONS— $C' = C - \{copy\};$
 SIGCONDITIONS— $card(\{x \mid book_id'(x) = book_id(copy)\}) = 0$
 $\rightarrow (Drop_title(book_id(copy)))ON;$

ENDEVENT;

EVENT *Check~out*(*copy*, *borr*: *B*);

PRECONDITIONS— $member(copy, C);$
 $not(borrowed(copy));$
 $books_out(borr) < limit;$

MAPCONDITIONS— $borrowed'(copy) = true;$
 $last_out'(copy) = borr;$
 $books_out'(borr) = books_out(borr) + 1;$

ENDEVENT;

EVENT *Check~in*(*copy*, *borr*: *B*);

PRECONDITIONS— $member(copy, C);$
 $last_out(copy) = borr;$

MAPCONDITIONS— $borrowed'(copy) = false;$
 $books_out'(borr) = books_out(borr) - 1;$

ENDEVENT;

TRANSACTION:

@ (T_min . now): ON(*Catalog~copy*(*newcopy*, *book*))OFF: *Add~copy*(*newcopy*, *book*);

ENDTRANSACTION;

ENDSEGMENT;

The answers to queries are simply values of functions for given arguments. Thus *last~out*(*copy*) returns the last borrower of a copy. However, to determine the set of books checked out to borrower *x*, we need a defined function

$out_set: B \rightarrow C-set: out_set(x) =$
 $\{y \mid last_out(y) = x \wedge borrowed(y)\}.$

Note now that function *out~set* makes the explicit map *books~out* redundant:

$books_out : B \rightarrow Integer : books_out(x) = card(out_set(x)).$

Queries regarding books by subject matter or by author can be given several interpretations. Under one interpretation the answer should be a listing of titles, irrespective of whether any copies of a particular title are indeed currently in the library. Under a second interpretation all currently available copies should be listed. Actually the most useful response seems to be a listing of all titles of which at least one copy is currently available:

$titles_by_topic : Area \rightarrow T-set:$

$$titles_by_topic(x) = \{y \mid member(x, subjects(y)) \wedge member(y, INCAT) \wedge \\ Allop(\vee; \{ not(borrowed(z)) \mid book_id(z) = y \})\}.$$

The definition of $titles_by_author$ is analogous.

Let us now consider authorization. Evaluation of functions $titles_by_topic$ and $titles_by_author$ can be requested by everyone. However, evaluation of $books_out(x)$ can be requested only by a librarian or by borrower x . This restriction takes care of itself if the identifiers of borrowers are known only to borrowers themselves and to the library staff, i.e., one must know x to access $books_out(x)$. Only librarians may initiate events.

SF Specification: An Elevator Controller

Alfs Berztiss
University of Pittsburgh

This is another example from the Fourth International Workshop on Software Specification and Design, also solved in set-function notation. This problem is more complicated than the library problem.

SF SPECIFICATION: AN ELEVATOR CONTROLLER

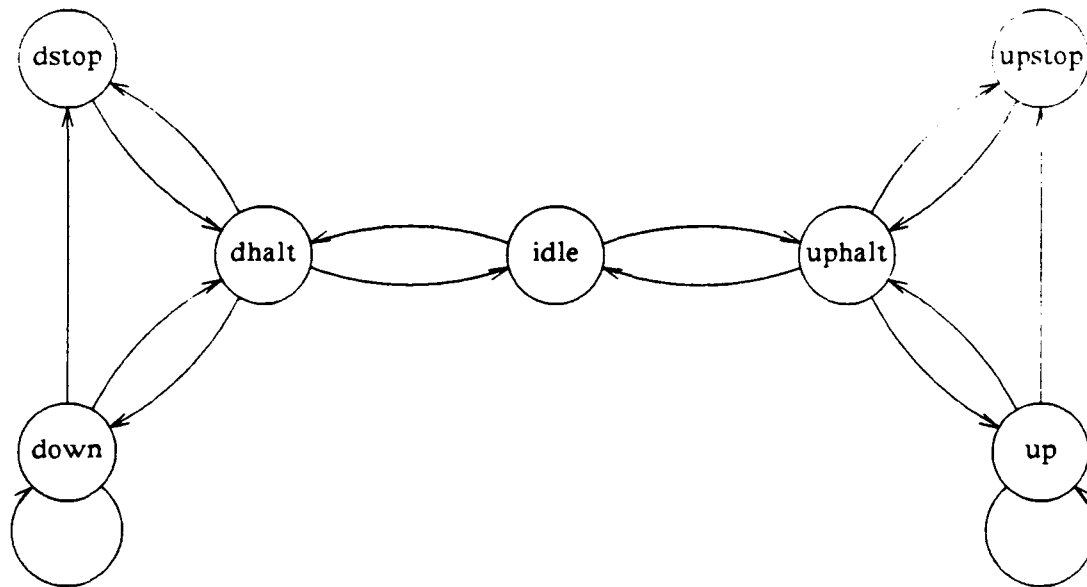
The elevator controller is the third problem selected for study at the Fourth International Workshop on Software Specification and design. Here the specification is to indicate the responses of a system to user requests. Each elevator has a set of buttons, one for each floor. They light up when pressed, and the elevator is then to visit the indicated floor. The light goes off when the elevator halts at this floor. The lit-up buttons define an agenda for the elevator. No floor can be added to the agenda that is not in the current direction of travel. All lights go off when a sensor determines that the elevator is empty. An elevator with an empty agenda is in an idle state.

Floors have outside buttons, to request up- or down-elevators. An outside button also lights up when pressed, and the light goes off when an elevator going in the appropriate direction visits this floor. The system should give equal priority to each outside request.

Each elevator has a stop button. Pushing in of this button causes the elevator to go out of service if it is at a floor, or to go out of service after reaching the next floor if it is between floors. The stop is terminated by pulling out the stop button.

We propose two segments, *Elevator* and *Dispatcher*. Suppose the system consists of k elevators. The elevator segment is to specify the operation of one such elevator in terms of the agenda. At implementation time k processes will be instantiated from this specification. The dispatcher segment is to look after the servicing of outside requests. The dispatcher would add floors to the agendas of the individual elevators, and send idle elevators to holding floors in order to provide efficient service. The dispatcher operation would be determined by a complicated scheduling algorithm, but the development of algorithms is a function of design rather than of specification. In other words, the purpose of specification is to indicate what a system is to accomplish, not how it is to do this. Consequently we can

define *Elevator* fully, but as regards *Dispatcher*, can do no more than pass through to the design stage the requirement that the system of elevators be fair.



The states of an elevator

The operation of an elevator can be looked at as transition between states, as shown in the figure above. When the elevator is in motion, it is either in an *up* or in a *down* state. When it halts at a floor, there is a transition to an *uphalt* or a *dhalt* state, respectively. From a halt state it can resume motion, become idle, or be stopped. The elevator can also be stopped, i.e., taken out of action, when it is in motion. There are two stop states, *upstop* and *dstop*, because there are two possible states of motion that the elevator can attain on resuming motion after passing into a halt state from a stop state. The elevator is taken out of the idle state by the dispatcher.

Actually the dispatcher can influence the operation of an elevator in three ways: an idle elevator may be moved to a particular floor; an idle elevator may be brought out of the idle state; an elevator in the up or down state or in one of the halt states may have a floor added to its agenda. The respective signals are *Move~idle*, *Activate~elevator*, *Add~to~agenda*.

Another way of looking at the actions of an elevator is in terms of sequences of events. Such sequences can be regarded as processes. The mechanism for defining a process is provided by SF signals, e.g., event A sets flag F, and a responder transaction initiates event B on account of flag F having been set. Such interleaving of events and transactions suggests the use of Petri nets to model the sequencing of events. This will be discussed further on.

The methodology used in the specification of the elevator differs in some ways from that used in the specification of the library system. One difference is that we introduce a special class of functions that we call *sensors*. They are actually devices that supply a value on demand. One sensor (*nullweight*) indicates whether the elevator is empty; another (*floor~now*) indicates the floor at which the elevator is currently located. There is also the *Next~floor~sensor*, which is a signal that is set whenever the elevator passes a special sensor in its travel between any two floors. This signal initiates an event (*Passing~sensor*) that determines whether the elevator is to continue in motion or is to halt at the next floor.

Continuation of motion or halting is regulated by two signals that enable segment *Elevator* to communicate with the mechanical controls of the elevator, namely the signals *Motion~up* and *Motion~down*. Very similar is the signal *Door~open*: it causes the elevator door to open when it is on, and the elevator door to close when it is off. The elevator is stopped by means of event *Stop~elevator*, which is initiated by the pressing of a stop-button inside the elevator. This event causes an alarm to sound (by means of the signal *Alarm*). The reciprocal event *Reactivate~elevator* stops the alarm. There are also signals to control the illumination of floor indicators: *Light*, one for each floor selection button within the elevator, and *Uplight* and *Dlight*, which are two buttons by which elevator service for going up or going down can be requested from the outside (of course the bottom and top floors have just a single button). We refer to these signals collectively as *mechanisms*.

SEGMENT *Elevator*;

IMPORTED SIGNALS *Activate~elevator*, *Add~to~agenda*, *Move~idle*;

SENSOR SIGNALS *Next~floor~sensor*;

MECHANISMS *Door~open*, *Alarm*, *Light*, *Uplight*, *Dlight*, *Motion~up*, *Motion~down*;

IMPORTED TYPE *Time*: *T* ENDTYPE;

IMPORTED TYPE *Time~interval*: *TI* ENDTYPE;

TYPE *Elevator*: *E*;

SECONDARY SETS— $S = \{\text{"idle"}, \text{"up"}, \text{"uphalt"}, \text{"upstop"}, \text{"down"}, \text{"dhalt"}, \text{"dstop"}\}$;

$\text{Floor} : F = \text{Integer}$;

FUNCTIONS— $\text{state} : E \rightarrow S$;

$\text{lowfloor} : E \rightarrow F$;

$\text{highfloor} : E \rightarrow F$;

$\text{clock} : E \rightarrow T$;

$\text{delay} : E \rightarrow TI$;

$\text{agenda} : E \times F \rightarrow \text{Boolean} \quad (\text{false})$;

SENSORS— $\text{floor~now} : E \rightarrow F$;

$\text{nullweight} : E \rightarrow \text{Boolean}$;

ENDTYPE;

EVENT *Initialize~elevator*(*e*; *low*, *high*: *F*; *interval*: *TI*);

(* Parameter *interval* indicates the time for which the elevator door is to be kept open after it was last opened or a person stepped through it. *)

MAPCONDITIONS— $\text{state}'(e) = \text{"idle"}$;

$\text{lowfloor}'(e) = \text{low}$;

$\text{highfloor}'(e) = \text{high}$;

$\text{delay}'(e) = \text{interval}$;

ENDEVENT;

INTERNAL EVENT *Activate~elevator*(*e*; *x*: *S*);

(* Initiated by the dispatcher via signal *Activate~elevator*. *)

MAPCONDITIONS— $\text{state}'(e) = x$;

$\text{clock}'(e) = T.\text{now}$;

SIGCONDITIONS— (Door \sim open)ON;
 $x = \text{"uphalt"} \rightarrow (\text{Uplight}(\text{floor} \sim \text{now}(e)))\text{OFF};$
 $x = \text{"dhalt"} \rightarrow (\text{Dlight}(\text{floor} \sim \text{now}(e)))\text{OFF};$
(Process \sim halt (e))ON;

ENDEVENT;

INTERNAL EVENT *Enter \sim halt*(e);

MAPCONDITIONS— agenda '(e , floor \sim now (e)) = false;
not(nullweight (e)) \rightarrow clock '(e) = T.now ;

SIGCONDITIONS— not(nullweight (e)) \rightarrow
BLOCK
(Door \sim open (e))ON;
state (e) = "uphalt" \rightarrow (Uplight (floor \sim now (e)))OFF;
state (e) = "dhalt" \rightarrow (Dlight (floor \sim now (e)))OFF;
(Light (e , floor \sim now (e)))OFF;
(Process \sim halt (e))ON;
ENDBLOCK;
nullweight (e) \rightarrow (Idle \sim elevator (e))ON;

ENDEVENT;

EVENT *Press \sim button*(e; floor: F);

(* Only floors in the direction of travel of the elevator may be added to the agenda. *)

PRECONDITIONS— state (e) = "up" \vee state (e) = "uphalt" \rightarrow floor > floor \sim now (e);
state (e) = "down" \vee state (e) = "dhalt" \rightarrow floor < floor \sim now (e);
not(member (state (e) , {"idle", "upstop", "dstop"}));

MAPCONDITIONS— agenda '(e , floor) = true;

SIGCONDITIONS— (Light (e , floor))ON;

ENDEVENT;

INTERNAL EVENT *Add \sim to \sim agenda*(e; floor: F);

(* Initiated by the dispatcher. *)

MAPCONDITIONS— agenda '(e , floor) = true;

ENDEVENT;

INTERNAL EVENT *Process*halt(*e*);

SIGCONDITIONS— $Allop(\wedge; \{ \text{not}(\text{agenda}(e, x)) \mid \text{lowfloor}(e) \leq x \leq \text{highfloor}(e) \}) \rightarrow$
 $(\text{Idle} \sim \text{elevator}(e))\text{ON};$
 $Allop(\vee; \{ \text{agenda}(e, x) \mid \text{lowfloor}(e) \leq x \leq \text{highfloor}(e) \}) \rightarrow$
 $(\text{Set} \sim \text{in} \sim \text{motion}(e))\text{ON};$

ENDEVENT;

INTERNAL EVENT *Set*inmotion(*e*);

MAPCONDITIONS— $\text{state}(e) = \text{"uphalt"} \rightarrow \text{state}'(e) = \text{"up"};$
 $\text{state}(e) = \text{"dhalt"} \rightarrow \text{state}'(e) = \text{"down"};$
SIGCONDITIONS— $(\text{Door} \sim \text{open}(e))\text{OFF};$
 $\text{state}(e) = \text{"uphalt"} \rightarrow (\text{Motion} \sim \text{up}(e))\text{ON};$
 $\text{state}(e) = \text{"dhalt"} \rightarrow (\text{Motion} \sim \text{down}(e))\text{ON};$

ENDEVENT;

INTERNAL EVENT *Passing*sensor(*e*);

MAPCONDITIONS— $\text{agenda}(e, \text{floor} \sim \text{now}(e) + 1) \rightarrow \text{state}'(e) = \text{"uphalt"};$
 $\text{agenda}(e, \text{floor} \sim \text{now}(e) - 1) \rightarrow \text{state}'(e) = \text{"dhalt"};$
SIGCONDITIONS— $\text{agenda}(e, \text{floor} \sim \text{now}(e) + 1) \rightarrow (\text{Motion} \sim \text{up}(e))\text{OFF};$
 $\text{agenda}(e, \text{floor} \sim \text{now}(e) - 1) \rightarrow (\text{Motion} \sim \text{down}(e))\text{OFF};$
 $\text{agenda}(e, \text{floor} \sim \text{now}(e) + 1) \vee \text{agenda}(e, \text{floor} \sim \text{now}(e) - 1) \rightarrow$
 $(\text{Enter} \sim \text{halt}(e))\text{ON};$

ENDEVENT;

EVENT *Stop*elevator(*e*);

MAPCONDITIONS— $\text{state}(e) = \text{"down"} \vee \text{state}(e) = \text{"dhalt"} \rightarrow$
 $\text{state}'(e) = \text{"dstop"};$
 $\text{state}(e) = \text{"up"} \vee \text{state}(e) = \text{"uphalt"} \rightarrow$
 $\text{state}'(e) = \text{"upstop"};$
SIGCONDITIONS— $\text{state}(e) = \text{"down"} \rightarrow (\text{Motion} \sim \text{down}(e))\text{OFF};$
 $\text{state}(e) = \text{"up"} \rightarrow (\text{Motion} \sim \text{up}(e))\text{OFF};$
 $(\text{Alarm}(e))\text{ON};$
 $(\text{Door} \sim \text{open}(e))\text{ON};$

ENDEVENT;

EVENT *Reactivate*[~]*elevator*(*e*):

MAPCONDITIONS— *state* '(*e*) = "upstop" → *state* (*e*) = "uphalt";

state '(*e*) = "dstop" → *state* (*e*) = "dhalt";

SIGCONDITIONS— (*Alarm* (*e*))OFF;

(*Enter* [~]*halt* (*e*))ON;

ENDEVENT;

INTERNAL EVENT *Idle*[~]*elevator*(*e*):

MAPCONDITIONS— *state* '(*e*) = "idle";

Allop (∧; {*not*(*agenda* '(*e*, *x*)) | *lowfloor* (*e*) ≤ *x* ≤ *highfloor* (*e*)});

SIGCONDITIONS— *Allop* (∧; {(*Light* (*e*, *x*))OFF | *lowfloor* (*e*) ≤ *x* ≤ *highfloor* (*e*)});

(*Door* [~]*open* (*e*))OFF;

ENDEVENT;

INTERNAL EVENT *Move*[~]*idle*(*e*; *floor*: *F*);

(* Initiated by the dispatcher. *)

MAPCONDITIONS— *agenda* '(*e*, *floor*) = true;

floor > *floor* [~]*now* (*e*) → *state* '(*e*) = "uphalt".

floor < *floor* [~]*now* (*e*) → *state* '(*e*) = "dhalt".

SIGCONDITIONS— (*Set* [~]*in*[~]*motion* (*e*))ON;

ENDEVENT;

EVENT *Update*[~]*clock*(*e*):

(* Initiated by breaking a light beam across the door of the elevator or by some similar device. *)

MAPCONDITIONS— *clock* '(*e*) = *T.now* ;

ENDEVENT;

EVENT *Open*[~]*door*(*e*):

(* This event is required for people to get out who somehow find themselves in an idle elevator. Raising the flag *Process*[~]*halt* ensures that the opened door will ultimately close again. *)

MAPCONDITIONS— *clock'(e) = T.now;*
SIGCONDITIONS— (*Door ~open(e)*)ON;
 (*Process ~halt(e)*)ON;

ENDEVENT;

TRANSACTION;

@(*T.now*): ON(*Activate ~elevator(e)*)OFF: *Activate ~elevator(e)*;

ENDTRANSACTION;

TRANSACTION;

@(*T.now*): ON(*Add ~to ~agenda(e, floor)*)OFF: *Add ~to ~agenda(e, floor)*;

ENDTRANSACTION;

TRANSACTION;

@(*T.now*): ON(*Move ~idle(e, floor)*)OFF: *Move ~idle(e, floor)*;

ENDTRANSACTION;

TRANSACTION;

@(*T.now*): ON(*Enter ~halt(e)*)OFF: *Enter ~halt(e)*;

ENDTRANSACTION;

TRANSACTION;

(* The delay is to give passengers time to press destination buttons. *)

@(*clock(e) + delay(e)*): ON(*Process ~halt(e)*)OFF: *Process ~halt(e)*;

ENDTRANSACTION;

TRANSACTION;

@(*T.now*): ON(*Set ~in ~motion(e)*)OFF: *Set ~in ~motion(e)*;

ENDTRANSACTION;

TRANSACTION:

@ (*T.now*): ON(*Next ~ floor ~ sensor (e)*)OFF: *Passing ~ sensor (e)*;

ENDTRANSACTION:

TRANSACTION:

@ (*T.now*): ON(*Idle ~ elevator (e)*)OFF: *Idle ~ elevator (e)*;

ENDTRANSACTION:

ENDSEGMENT:

The Specification Process

Alfs Berztiss
University of Pittsburgh

The process of creating specifications, especially in the set-function methodology, is described. Note that the elevator problem (previous section) is discussed at the end of this section.

THE SPECIFICATION PROCESS

SEGMENTATION OF SF SPECIFICATIONS

The initial task in the specification process is to define the segments. Discussions with clients and within specification groups establish a common vocabulary. For the most part data types derive from nouns in the vocabulary, events from verbs. Functions are defined in anticipation of the queries that will be put to the system or of the support needs for the control activities of the system. (The information needed to evaluate preconditions of events can be regarded as provided by internal queries.) At this stage no thought should be given to responders or signals. Nevertheless, the initial scheme will become modified a few times.

Alternatives will have to be weighed one against another. For example, a withdrawal from an account may be considered as an event that modifies functions belonging to segment *account*. Alternatively, *withdrawal* could itself be a segment. There would still be a withdrawal event, but this event would modify functions of segment *withdrawal*. Of course, withdrawals also affect balances in accounts, but the balance adjustments can be accomplished by means of signals that initiate internal events in *account*.

It is important to realize that there is no "best" solution, although we do recommend that each segment be identified with a data type. The first factor to affect segmentation is the client's viewpoint. If the primary purpose of the banking system is to provide information regarding accounts, then there may not be a need for a separate withdrawals segment. But the need may exist if the primary purpose is to control the processing of withdrawals. In other words, the segmentation should correspond to a partitioning of the system that seems natural to the client. Therefore it is necessary to hold extensive consultations with representatives of the client. It is essential that the segment structure be found acceptable

by these representatives before further work is undertaken. Unfortunately, a realistic client-specifier interaction is difficult to provide for student projects.

Second, segmentation assists in the distribution of labor. If a team of four is to specify an elevator controller, then a separation of the system into four equal segments may be appropriate, but equality (of size or difficulty) of segments is hard to achieve. A group of four students that worked on the specification of an elevator used segments *get~calls* (to collect destination indications from within and calls for service from without the elevator), *add~floors* (to set up an agenda), *move~elevator* (to see to the actual movement of the elevator, both when it is responding to users and when it is idle), *dispatcher* (to switch agendas), *abnormal~stop* (to deal with the pressing of the stop button), and *power~on~off* (to deal with power failures).

Some of these segments consist merely of events and the responder, which goes against our earlier recommendation that segments be data types. Although the approach was justified here, in the long run excessive segmentation causes a heavy traffic of messages between segments. Therefore, after the segments have been developed by members of the specification team, and the initial design tested by some static analysis technique, such as a walkthrough, segments should be amalgamated. This is a very simple process in that the existing messages will still be needed, but they will now be passed along internally between components of the same segment.

Other important decisions relate to type hierarchies. How should the ISA facility be used? When is it better to have a hierarchy? When is it better to have independent segments? For a while we toyed with the idea that the titles and copies of the library system should somehow be made into a hierarchy, but finally rejected the idea as counterintuitive. The statement " $X \text{ ISA } Y$ " only makes sense for X a subset of Y . Thus, in the boat example, the set of boats in the system is a proper subset of all objects available for hire, and this set of boats is also a subset of all water vehicles. Actually we could have refined the second subset relation by stating that our set of boats is a subset of the set of all boats, with the

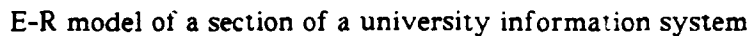
latter being declared a subset of all water vehicles.

The set of titles, on the other hand, defines a partition superimposed on the set of copies. Whereas every boat is a water vehicle, one copy corresponds to one title, and another copy corresponds to another title. The ISA should not be used to define aggregations either. For example, a car may be composed of four wheels, a chassis, etc. This means neither that "*car* ISA *wheel*", nor that "*wheel* ISA *car*".

Once the precise structure of the segments has been fixed, the pre-, set-, and mapconditions of the events can be filled in. To avoid oversights, a list of all the functions of the data type should be consulted whenever a new event is being defined, to check that all effects of the event are indeed being considered. At this stage the need for additional functions may become felt.

An important tool for identifying all the objects or entities that are to be the concern of an information-control system, and for relating these entities to each other is Chen's Entity-Relationship (E-R) approach. It is primarily a graphical device for displaying a static model of an enterprise. There are two kind of boxes: rectangles for entity sets, and diamonds for relationship sets. Boxes are linked by undirected lines, and the two boxes linked by a line differ in kind. Further, both entity sets and relationship sets may have attributes. An attribute of a set is indicated by an ellipse, and there is an arrow from the box representing the set to the ellipse.

As an illustration we show an E-R diagram for an information system that relates students, courses, and instructors. There are two relationships, *Takes* and *Gives*. The labels *N* and *M* mean that *Takes* is a many-to-many (*N*-to-*M*) relation, and labels *N* and 1 mean that *Gives* is many-to-one from *Course* to *Instructor* (or one-to-many from *Instructor* to *Course*). We also show some of the attributes of *Course*, and an attribute of *Gives*. The latter indicates whether a given instructor teaches a given course on a regular basis or under some special arrangement.



TYPE *Student*: S;

ENDTYPE;

47

```

      . . . . .
      . . . . . (Further attribute functions)
      . . . . .

ENDTYPE;

TYPE Instructor: F;

  SECONDARYSETS- Teaching~status: TS = {"REG", "SPEC"};

  FUNCTIONS-   gives:       $F \rightarrow C$  -set: gives(x) =
                                     {y | instructor of (y) = x};
               status:  $F \times C \rightarrow TS$ ;
               . . . . .
               . . . . . (Attribute functions)
               . . . . .

ENDTYPE;

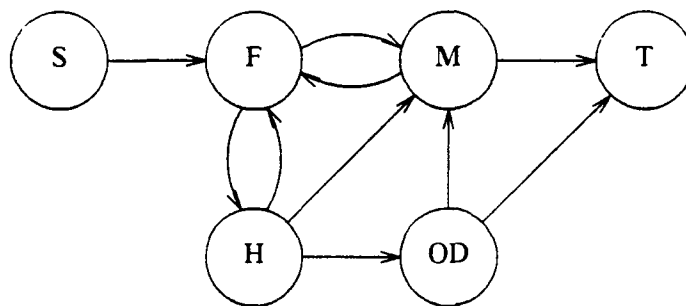
```

The E-R philosophy is to differentiate between relations and attributes. Relations are between entities that belong to the E-R model; attributes are maps from an entity or a relation to a data type that is external to the model. However, with the separation of the model itself into segments such a distinction becomes rather arbitrary and artificial. Furthermore, the vocabulary is increased unnecessarily. Nevertheless, the development of an E-R model of an information system is certain to contribute to a better understanding of the interrelation of the elements of the information system, and is therefore recommended as a preliminary to SF specification.

THE DYNAMIC COMPONENT OF SF SPECIFICATIONS

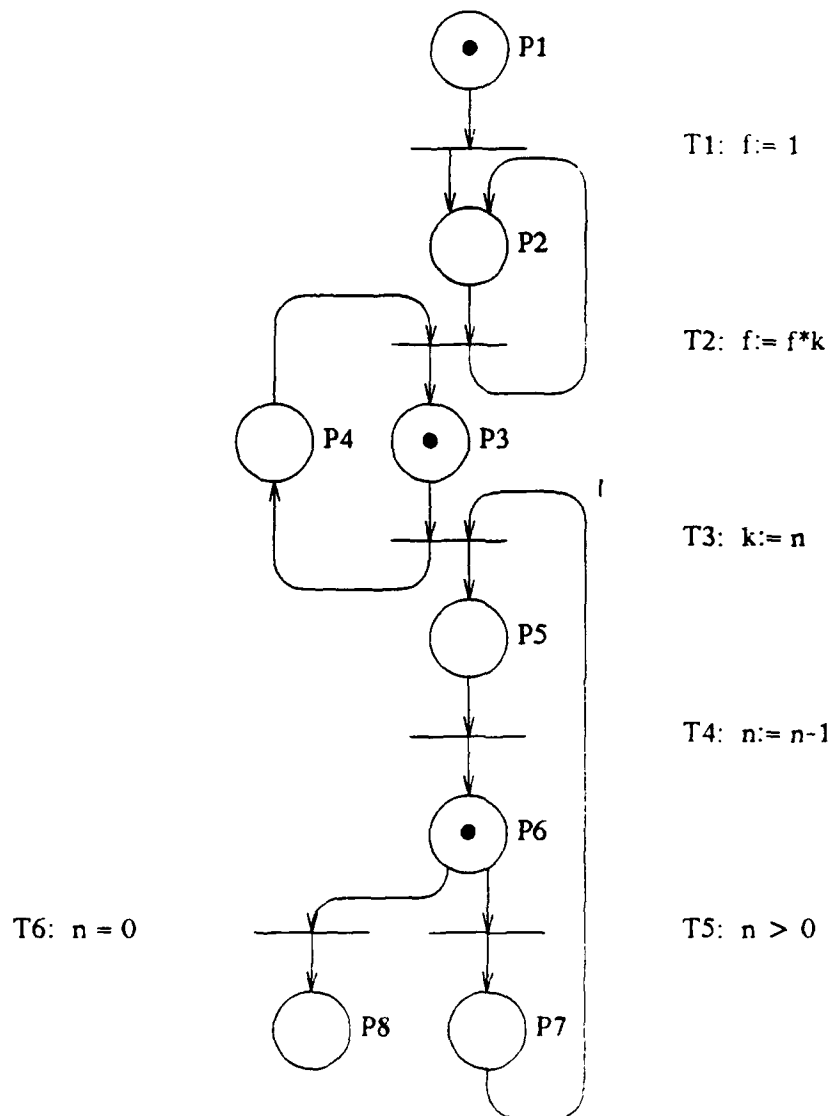
The final stage in SF specification is the addition of the dynamic (or time-dependent, or control) component. This consists of sigconditions, the responder, and internal events. In some cases the dynamic component is a fairly minor addition, such as in the library example. Surprisingly, in all our experience, the addition of the dynamic component has required no or very minor changes to the static part of the specification.

We have found state-transition diagrams and Petri nets of great assistance in the writing of SF specifications. An example of a state-transition diagram has already been given for the elevator problem. For the boat-hire problem the appropriate state-transition diagram is as follows:



A (place-transition) Petri net consists of a digraph and a dynamically changing marking pattern. The digraph consists of two disjoint sets of nodes, places P and transitions T and a flow relation $F \subseteq (P \times T) \cup (T \times P)$. Given transition t , the set of places $\{p \mid \langle p, t \rangle \in F\}$ is the preset of t and the set of places $\{p \mid \langle t, p \rangle \in F\}$ is the postset of t . Presets and postsets of places are defined analogously. Places are usually represented by circles, transitions by bars. Each place has associated with it a class of token types. Tokens of different types may be distinguished by drawing them in different colors or by use of different symbols. An initial marking provides each place with zero or more tokens of each of the types associated with this place, and the net is then in its initial state. The net changes states by firings. A firing of a transition is enabled if each place in its preset holds at least one token of each of the types associated with that place. The result of the firing is twofold. First, for each place in the preset, one token of each of the types associated with that place is removed. Second, for each place in the postset, a token of each type associated with this place is added. Tokens may also be added during the running of a Petri-net process as inputs. In the examples to be discussed here all tokens will be of the same type. Note that we shall use Petri nets primarily as a graphical tool. An investigation of the uses

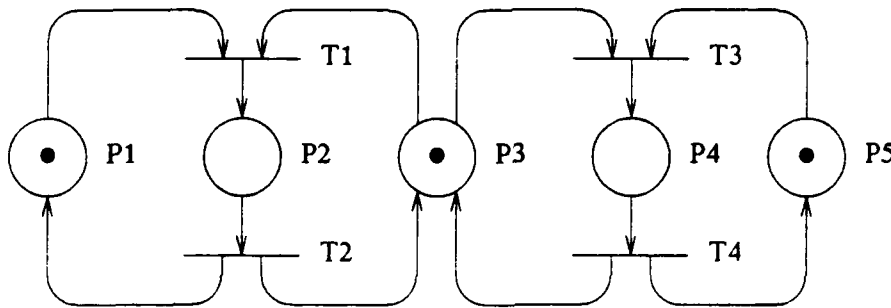
of the formal theory underlying Petri nets in the analysis of control systems is outside our scope.



Petri net for the computation of the factorial

Let us start with a very simple computation--the evaluation of $factorial(n)$, where n is a nonzero integer. Our first example of a Petri net is for this computation, and it is shown with its initial marking. Transitions represent steps in the computation, and the movement of tokens between places seems to it that these computational steps are properly

sequenced. If a transition is annotated with a predicate, as T5 and T6 are in the figure, then the normal firing rule is augmented with the requirement that the predicate be true. Suppose $n = 0$. Then T1 and T6 can fire. When they do, tokens move into P2 and P8, and the process stops with $f = 1$ because no further firings can take place. On the other hand, if n is greater than zero, then T1 fires as before, but now it is T5 that fires at the lower end. This moves a token into P7, and T3 can fire. Firing of T3 moves tokens into P4 and P5, so that T2 and T4 can fire next. Note the parallelism: P2, T2, P3, and P4 represent one process, the building up of the factorial by the multiplication at T2; P5, T4, P6, T5, and P7 represent another process, the adjustment of n at T4. The point of contact of the two processes is T3, where the current value of n is assigned to k . An alternative would be to pass the current value of n in a message from process to process at T3. In any case, the design of the network and the initial marking make sure that the two processes do not get out of step.



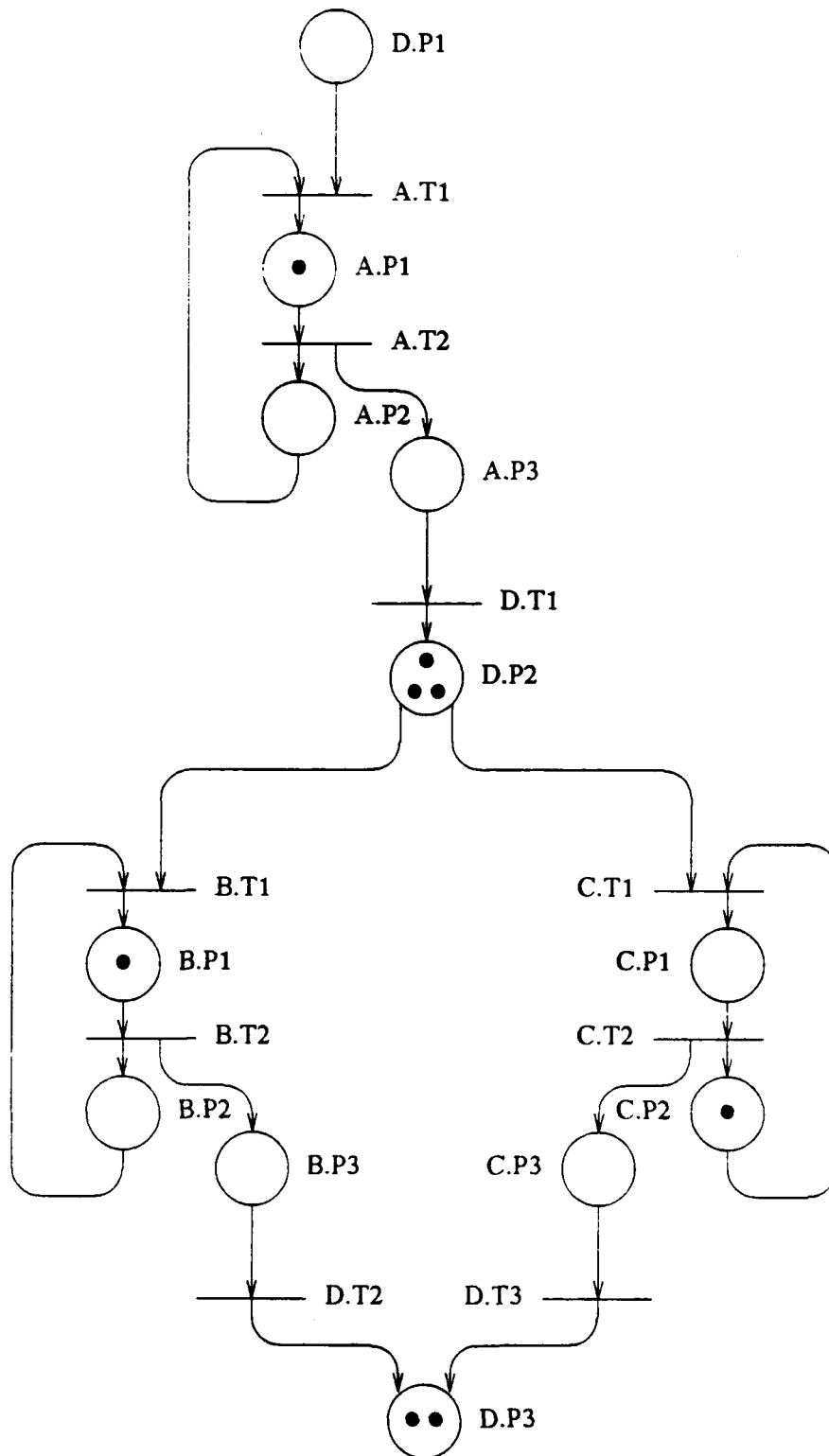
Petri net representation of mutual exclusion

Next we consider two processes that are to have access to a particular resource, but never both at the same time. This is known as *mutual exclusion*. In the figure, let a token in P2 represent access to the resource by process A, and a token in P4 access by process B. Under the marking as shown, both T1 and T3 are enabled to fire. Suppose T1 fires. This removes tokens from P1 and P3 and places a token in P2. While this token remains in P2, transition T3 cannot fire, i.e., while there is a token in P2 there cannot be a token in P4, and

vice versa.

In the specification of systems that are primarily control systems a Petri net may be used from the very start to define the structure of the system. We show a net that represents a manufacturing system consisting of one producer and two consumers. The "producer" *A* consists of places *P1*, *P2*, and *P3*, and transitions *T1* and *T2*. "Consumers" *B* and *C* have the same configuration as the producer. For example, the producer could assemble toasters, and the consumers package the toasters. The producer generates at *P1* objects, which are represented by tokens. Firing of *T2* sends one token to *P2* to keep the process going, and another, which represents the generated object, to *P3*. The generation of an object is triggered by the placing of a token in *D.P1* (an external input), and the objects finally go to place *D.P2*, which acts as a buffer. Tokens are taken off *D.P2* by either *B.T1* or *C.T1*, the entry points to the consumer processes. The actual processing occurs at *B.P1* and *C.P1*, and the finished goods end up in *D.P3*, via *B.P3* or *C.P3*. The initial marking consists of one token each at *A.P2*, *B.P2*, and *C.P2*.

The translation of the Petri net into SF is schematic in that we ignore anything to do with the data base, i.e., preconditions, setconditions, and mapconditions. It must be understood that in SF the time at which a transition fires is explicitly defined, and that an event triggered by a transition only takes place if its preconditions are satisfied. We let each of processes *A*, *B*, *C* of the figure be an SF segment. Consider the producer process, where we assume that the actual time to produce an object is 20 minutes. The schematic specification of segment *A* now follows. There signal *T1 ~start* is set outside the segment, and signal *Process ~complete* triggers an event outside of segment *A*. Event *A.P3* (and transition *D.T1* appear superfluous), but they are needed to support an SF convention that no event may be initiated directly from another segment. This contributes to the ease of developing segments independently of each other. Hence we need a transition within segment *D* to initiate *D.P2* (transition *D.T1*), and an event in *A* that is to trigger this transition (event *A.P3*).



Petri net of a manufacturing system

SEGMENT A:

IMPORTED SIGNALS $T1 \sim start$:

EXPORTED SIGNALS $Process \sim complete$:

INTERNAL EVENT $P1$:

SIGCONDITIONS— ($T2 \sim trigger$) ON:

ENDEVENT:

INTERNAL EVENT $P2$:

SIGCONDITIONS— ($T1 \sim trigger$) ON:

ENDEVENT:

INTERNAL EVENT $P3$:

SIGCONDITIONS— ($Process \sim complete$) ON:

ENDEVENT:

TRANSACTION $T1$:

@ ($T \sim Now$): AND(ON($T1 \sim start$) OFF, ON($T1 \sim trigger$) OFF): $P1$:

ENDTRANSACTION:

TRANSACTION $T2$:

@ ($T \sim Delay(20, T \sim Now)$): ON($T2 \sim trigger$) OFF: $P2$; $P3$:

ENDTRANSACTION:

ENDSEGMENT:

The delay of 20 minutes represents the time required to produce an object. If signals are combined by an AND, they must both be in the ON-state for anything to happen, e.g., if $T1 \sim start$ alone is on when $T1$ is processed, then $T1 \sim start$ does not get turned off. The specifications of segments B and C have exactly the same form as that of A . The only difference is in transaction $T2$: this transaction is likely to have shorter delays in B and C . The modularization of the system has been carried out in such a way that the specifications of segments A , B , and C would not have to be changed if they were to be used in some other context. Here they are coordinated by segment D , and a schematic specification of D

now follows.

SEGMENT *D*;

IMPORTED SIGNALS *A.Process ~complete* , *B.Process ~complete* , *C.Process ~complete* ;

EXPORTED SIGNALS *A.T1 ~start* , *B.T1 ~start* , *C.T1 ~start* , *P3 ~inventory* ;

EVENT *P1* ;

SIGCONDITIONS— (*A.T1 ~start*)ADD;

ENDEVENT;

INTERNAL EVENT *P2* ;

SIGCONDITIONS— AND(OFF(*B.T1 ~start*)ON, ON(*P2 ~inventory*)OFF);

AND(OFF(*C.T1 ~start*)ON, ON(*P2 ~inventory*)OFF);

ENDEVENT;

INTERNAL EVENT *P3* ;

ENDEVENT;

TRANSACTION *T1* ;

@ (*T ~Now*) : BLOCK

ON(*A.Process ~complete*)OFF: (*P2 ~inventory*)ADD;

OR(OFF(*B.T1 ~start*), OFF(*C.T1 ~start*)): *P2* ;

ENDBLOCK;

ENDTRANSACTION;

TRANSACTION *T2* ;

@ (*T ~Now*) : ON(*B.Process ~complete*)OFF: (*P3 ~inventory*)ADD;

ENDTRANSACTION;

TRANSACTION *T3* ;

@ (*T ~Now*) : ON(*C.Process ~complete*)OFF: (*P3 ~inventory*)ADD;

ENDTRANSACTION;

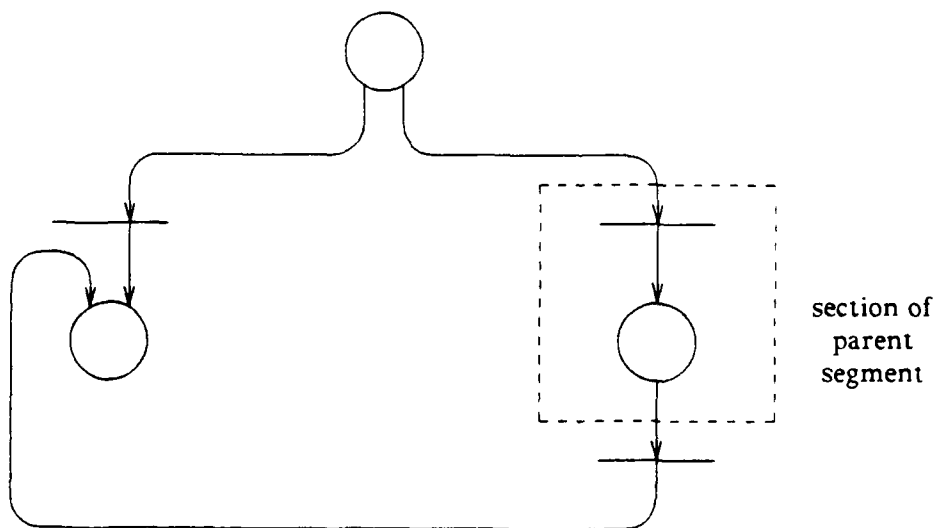
ENDSEGMENT;

A.T1 ~start , *P2 ~inventory* , and *P3 ~inventory* differ from other signals. They are

counters rather than flags. An ADD operation increments a count, and also identifies a signal as a counter. In this context $ON(Counter)$ is true if $Counter$ is non-zero, and $(Counter)OFF$ decrements $Counter$. The computation associated with the processing of a signal, such as $ON(P2 \sim inventory)OFF$, must be atomic. Otherwise both the $BT1 \sim start$ and $CT1 \sim start$ of our example could get set when there is only one item in the inventory. Event D.P3 is left totally unspecified. Another process could be fed by it, such as the distribution of finished products to sales points.

A problem that arises quite frequently has to do with the ISA. Suppose that referees form a subset of persons (asserted by means of an ISA), individual x is being selected as a referee, but individual x is not in the set of persons P . One approach is to test for membership of x in P , and simply not to proceed with the referee selection event if the test is not satisfied.

A better solution is to make referee selection a two-stage process. First test for membership in P . If the test is passed, proceed with referee selection (as an internal event). Otherwise raise a signal that prompts an event in the parent segment to add x to P . This event raises a signal that causes x to become a referee. The Petri net of this paradigm:

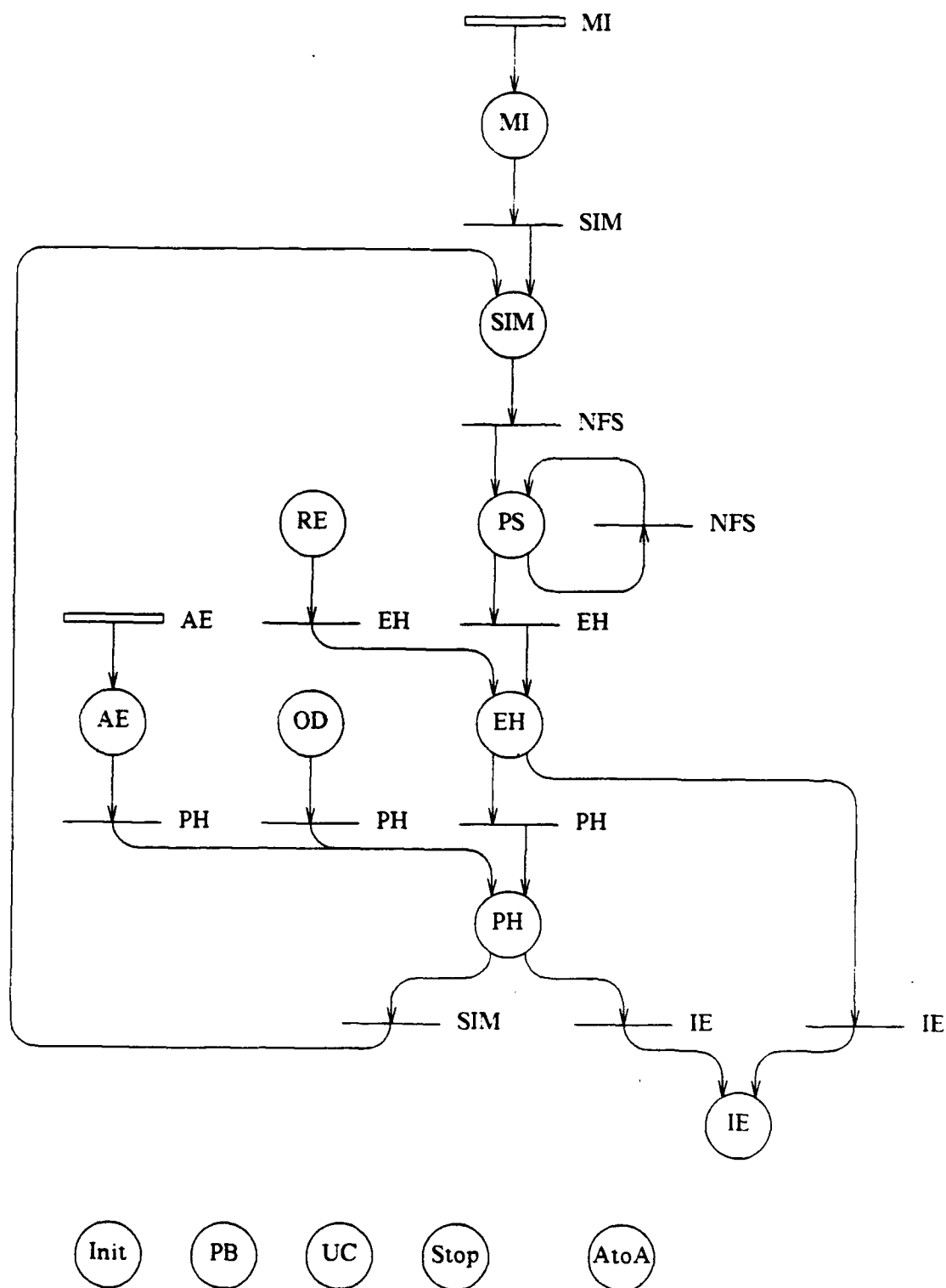


SPECIFICATION OF THE ELEVATOR

The requirements statement for the problem asks for the specification of a *system* of elevators. Initially we hoped to develop two segments--an *elevator* segment, and a *dispatcher* segment that was to decide which elevator was to respond to an outside call for service and was to move idle elevators to strategically picked holding floors. But the "specification" of the dispatcher then encroaches on design. An algorithm has to be selected that is based on queueing theory and scheduling theory, and the algorithm may have to be tuned on the basis of simulation experiments. We maintain is that a specification should be prealgorithmic, but this means that the dispatcher cannot be specified. Of course, if the requirements actually propose an algorithm, then the specification should reflect the characteristics of the algorithm. This was the case with the two-way merge. In the present instance, if an algorithm had been supplied for the dispatcher, then we could have written a specification, but we could not be expected to devise the algorithm.

As regards the elevator segment, we realized from the very start that we should consider state transitions. However, at first we had only the states *idle*, *halt*, and *move*. It did not take long to realize that the *halt* state could not cope with emergency stops, and a separate *stop* state was added. However, a whole day of false starts was wasted before the realization that states *move*, *halt*, and *stop* had to be split on the basis of whether the elevator movement is up or down.

This put us in a position to define external events, but when it came to the linking of events into processes, and the definition of internal events (eight of a total of fourteen), the interrelation of events and transactions was becoming so tangled that we had no overview of the total system. The Petri net representation showed below established the overview. In the diagram of the net we use abbreviations, as listed in the table on the page that follows the diagram of the net.



Petri net of an elevator

AE -	<i>Activate elevator</i>	PB -	<i>Press button</i>
AtoA -	<i>Add to agenda</i>	PH -	<i>Process halt</i>
EH -	<i>Enter halt</i>	PS -	<i>Passing sensor</i>
IE -	<i>Idle elevator</i>	RE -	<i>Reactivate elevator</i>
Init -	<i>Initialize elevator</i>	SIM -	<i>Set in motion</i>
MI -	<i>Move idle</i>	Stop -	<i>Stop elevator</i>
NFS -	<i>Next floor sensor</i>	UC -	<i>Update clock</i>
OD -	<i>Open door</i>		

As before, places stand for events, transitions for transactions. The transitions carry labels that refer to the signals processed by the transactions that these transitions represent. The broader bars indicate transactions that are initiated from the outside. The five place symbols that are not part of the main net represent events that do not directly cause other events; one of these, event *Add to agenda*, is initiated by the dispatcher.

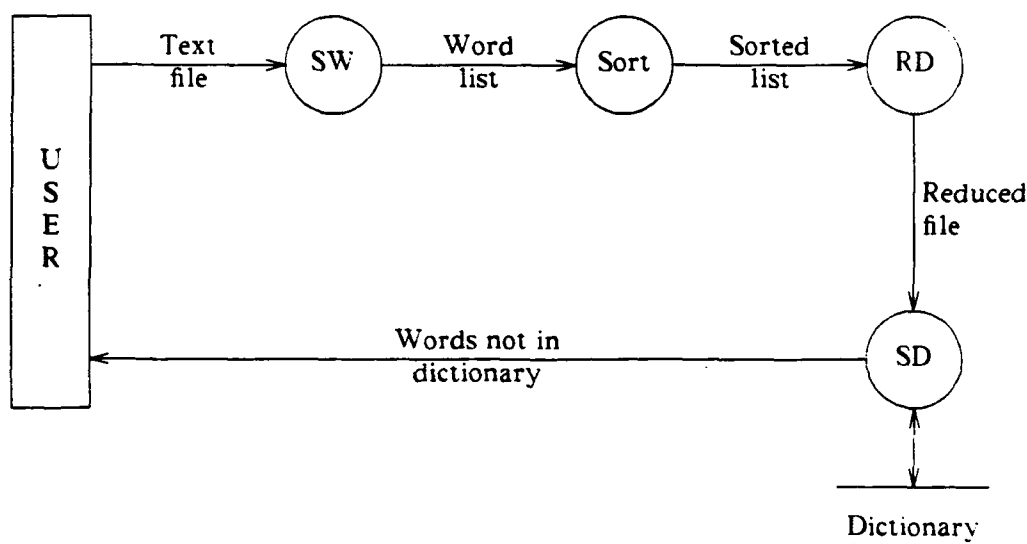
SPECIFICATION OF DATA TRANSFORMERS

Data flow diagrams can assist in the specification of data transformers. These diagrams form the basis of the "structured analysis" approach to development of systems. A data flow diagram is built up of components of four types: (1) Sources and sinks, which are external agencies that send data into the system or receive data from the system. They are represented by boxes. (2) Processes, which are represented by circles. (3) Files or data bases, represented by two parallel lines. (4) Data flows, which are represented by arcs linking pairs of objects of the first three types.

An information-control system is based on a centralized data base, and all "flow" relates just to the updating and interrogation of this data base. Consequently data flow diagrams are of little use in the specification of information-control systems. But the situation is quite different with data transformers. Here actual data objects are moved from

process to process, and each process carries out some transformation of these objects.

Our example is the data flow diagram of a spelling checker. A text file enters the system and the text is split into words at node SW. The word list that is the result of this operation is sorted, and process RD removes duplicates from the sorted list. At SD (Subtract Dictionary) this reduced list is compared against a stored dictionary, and words not found in the dictionary are presented to the user.



Data flow in a spelling checker

Actually, with this diagram we have moved well out of specifications and into software design. The indication that the word list produced from the text file should be sorted, and duplicates removed from it, is the selection of an algorithm. A specification should not go that far. Instead, for a specification we should regard the word list (call it W) and the dictionary (D) as sets, and the errata list is then the set difference $W-D$. How the set difference is obtained should be of no concern to the writer of the specifications.

Formal Specification Courses

Alfs Berztiss
University of Pittsburgh

Details of course organization are described. Examples are drawn from experiences at the University of Pittsburgh. Suggestions for exercises and projects are included.

FORMAL SPECIFICATION COURSES

COURSE OUTLINES

These outlines relate to courses dealing with formal specification offered by the Department of Computer Science, University of Pittsburgh. Course CS135 is taken by undergraduates, usually in their senior year. The prerequisite structure ensures that they have completed at least five computer science courses. CS231A is a graduate course, where the only prerequisite is admission to graduate study in computer science. At the University of Pittsburgh this essentially means that the student has an undergraduate major in computer science. Each lecture lasts 80 minutes.

CS135- SOFTWARE SYSTEMS DESIGN

Fall 1987/8

- 1 (Sep. 2): Introduction to course
- 2 (Sep. 9): Software life cycle
- 3 (Sep. 14): Components of computations
- 4 (Sep. 16): Principles of modularization
- 5 (Sep. 21): The SF (set-function) methodology
- 6 (Sep. 23): Specification: Boat hire
- 7 (Sep. 28): Specification: A library system
- 8 (Sep. 30): Initial discussion of the term project
- 9 (Oct. 5): Petri nets I
- 10 (Oct. 7): Petri nets II
- 11 (Oct. 12): Specification: An elevator
- 12 (Oct. 14): Specification: A text formatter
- 13 (Oct. 19): Testing of specifications

(Oct. 21): MIDTERM EXAMINATION (open book)

- 14 (Oct. 26): Site-related aspects of the SF methodology
- 15 (Oct. 28): Property inheritance and knowledge representation
- 16 (Nov. 2): Review of the SF methodology
- 17 (Nov. 4): More on testing
- 18 (Nov. 9): More on the project
- 19 (Nov. 11): Errors, uncertainties, exceptions
- 20 (Nov. 16): Entity-relationship model

- 21 (Nov. 18): Abstract data types I
- 22 (Nov. 23): Abstract data types II
- 23 (Nov. 30): Data types and generators
- 24 (Dec. 2): Programming with generators
- 25 (Dec. 7): From specifications to software I
- 26 (Dec. 9): From specifications to software II

(Dec.17): FINAL EXAMINATION (open book) 10:00-11:20

CS231A- SOFTWARE ENGINEERING: SPECIFICATION AND DESIGN

Winter 1987/8

- 1 (Jan. 7): Introduction to software engineering
- 2 (Jan.12): Software life cycle
- 3 (Jan.14): Components of computations
- 4 (Jan. 19): Abstract data types: introduction
- 5 (Jan. 21): Hierarchies and modules
- 6 (Jan. 26): The SF (set-function) methodology
- 7 (Jan.28): Specification: bank accounts
- 8 (Feb. 2): Specification: a library system
- 9 (Feb. 4): Discussion of the term project
- 10 (Feb. 9): Petri nets: introduction
- 11 (Feb.11): Petri nets and SF specification
- 12 (Feb.16): Petri nets in process analysis
- 13 (Feb.18): Specification: an elevator
- 14 (Feb.23): Testing of specifications
- 15 (Feb.25): Abstract data types I
- 16 (Mar. 1): Abstract data types II

(Mar. 3): MIDTERM EXAMINATION (open book)

- 17 (Mar.15): Review of the SF methodology
- 18 (Mar.17): Specifications: a text formatter and two-way merge
- 19 (Mar.22): Data types and generators
- 20 (Mar.24): Programming with generators
- 21 (Mar.29): The Larch approach to specifications
- 22 (Mar.31): Larch and CLU
- 23 (Apr. 5): The Z specification language
- 24 (Apr. 7): Z and specification of processes
- 25 (Apr.12): The PAISley and MSG.84 methodologies
- 26 (Apr.14): The Vienna Development Method in specification
- 27 (Apr.19): Sites and knowledge representation in SF
- 28 (Apr.21): From specifications to software by transformations: I
- 29 (Apr.26): From specifications to software by transformations: II

(Apr.28): FINAL EXAMINATION (open book)

PRACTICAL WORK

A course dealing with specification should be centered on a major group project, but several individual exercises are an essential preliminary to the group project. First, this allows the instructor to find out which specification principles and methodologies the instructor had not fully explained. Second, misconceptions by individual students can be corrected. Third, the instructor comes to understand the capabilities of the students. On an individual basis this helps during the selection of teams for the group project. More broadly, the level of difficulty of a project can be matched to the experience of the class.

As regards the group project, we have generally started out with four to five students to a specification team. A smaller group may become ineffective if a member drops out or fails to contribute adequately, and the duration of projects is too short to justify larger groups. Our standard practice has been to assign students to groups in alphabetical order. However, if chance puts too many weaker students in a group, some switching should be done. Otherwise the group may not get going. For greater realism, we intend to experiment with the transfer of stronger students from one group to another (that works on a different project) halfway through the exercise.

Our grading scheme has been to derive 50% of the total grade from examinations and 50% from individual assignments and the project, but at times we have dispensed with the midterm examination, in which case the project has carried 50%, assignments 25%, final examination 25%.

A project is first given an overall grade, for example along the lines

- Quality of the specification - 20 points
- Documentation - 10 points
- Project log - 5 points
- Validation (walkthroughs) - 10 points
- General presentation - 5 points

The overall grade may then be reduced (sometimes raised) for individual members of the team according to three criteria. First, each student is required to do a confidential rating of

all members of his or her team by indicating the percentage of the total effort contributed by each individual. Second, each group keeps a project log. The log provides a record of attendance at meetings and of individual assignments. Third, the quality of the work assigned to an individual is compared to the quality of the work by others in the group. However, group interaction tends to smooth out differences in quality of a specification, i.e., quality tends to be uniform across an entire specification document. Indeed, students working as a group influence each other in such a way that all members contribute equally, with the weaker students investing more time in the project. The exceptions are few, and the log and the student ratings identify them very well (a lowered score for just six of 42 students is typical).

A LOGBOOK: ELEVATOR SPECIFICATION

Meeting # 1
Feb. 2, 1987
6:30 Hillman Lib.

Absent: Nobody

Accomplished: Group members simply got acquainted and discussed possible meeting times. Monday evenings were discovered to be the best time to meet, and it was agreed upon by all members.

Meeting # 2
Feb 19, 1987
6:00 Hillman Lib.

Absent: Nobody

Accomplished: Basic setup of functional requirements was established. It was decided that two segments would be required; an operator, and a dispatcher. The imported types and signals were agreed upon and written. Each member was then assigned specific events and everybody agreed to write one version of the event Next_Move.

Channarasappa: Add_Call_To_Agenda
Move_Elevator

Correa: Delete_Floor_From_Agenda
Halt_At_A_Floor

Eaton: Move_Elevator
Stop_At_A_Floor

Fetsko: Add_Select_Floor_To_Agenda
Emergency_Stop

Meeting # 3
Feb. 23, 1987
6:00 Hillman Lib.

Absent: Nobody

Accomplished: Basic structure of operator and dispatcher put together. The assigned events were reworked at meeting so they all used similar terms and notations. The event Next_Move was assigned to all group members again since no workable solution could be found. The two main types: Elevator and Agenda, were then written and agreed upon.

Meeting # 4
Feb. 24, 1987
3:30 Alumni Lib.

Absent: Nobody

Accomplished: As a group effort, an ASM chart was setup for the event Next_Move. From this chart, we were able to find the signal and map conditions necessary to have a complete working version. Once written, The event was vigorously tested and subsequently found to be satisfactory.

Meeting # 5
March 2, 1987
6:00 Hillman Lib.

Absent: Nobody

Accomplished: All the events were tied together after some editions to them. The functions of each segment type were then written. And finally, a rough version of the functional requirements was completed. Each member was then assigned to type in his or her events into one account or file.

Meeting # 6
March 4, 1987
6:00 Hillman Lib.

Absent: Nobody

Accomplished: This meeting was more or less an initial debugging session. As a result, errors were detected in many areas of the requirements and fixed by the group.

Errors detected in:

Dispatcher:	imported signals and types
Type Agenda:	functions
Add_Select_Floor_To_Agenda:	mapconditions
Next_Move:	mapconditions
Operator:	imported signals
Type Elevator:	functions
Move_Elevator:	map conditions
Halt_At_A_Floor:	signal conditions

Meeting # 7
March 9, 1987
6:00 Forbes Quad.

Absent: Nobody

Accomplished: A format for the documentation was setup and agreed upon. Each member was then assigned to write an equal portion of it and type it in.

Channarasappa: Segment Operator
Type Elevator
Event Stop_At_A_Floor
Halt_At_A_Floor

Correa: Segment Dispatcher
Event Add_Select_Floor_To_Agenda
Event Move_Elevator
Event Emergency_stop

Eaton: Event Add_Call_To_Agenda
Event Delete_Floor_From_Agenda
Event Stop_Release

Fetsko: Introduction
Event Next_Move
Type Agenda

Meeting # 8
March 16, 1987
6:00 Hillman Lib.

Absent: Nobody

Accomplished: Material was passed out for walkthrough. A date, time, and location was setup at the Hillman Library on March 18, 1987 at 4:00 PM.

Meeting # 9
March 18, 1987
4:00 Hillman Lib.

Absent: Nobody

Accomplished: The walkthrough. Errors were detected in the event Emergency_Stop which required the addition of the event E_Stop. Errors were also detected in the segment operator.

Note: These errors were not fixed until the walkthrough was completed.

Meeting # 10
March 21, 1987
6:00 Hillman Lib.

Absent: Nobody

Accomplished: The final version was reviewed and accepted by all the group members. PROJECT COMPLETE!

FROM OTHER LOGBOOKS

Specification groups working on a student registration system were encouraged to interview university staff actually concerned with registration before requirements were defined. Only one member of the team was to meet with each official. Members of one group interviewed three university officials: the Director of the Undergraduate Programs Office in the Department of Computer Science (4 pages in the log), an official in Student Business Services (another 4 pages in the log), and a student advisor in the College of General Studies (2 pages in the log). This group maintained three logs: a chronological log, a walkthrough log, and a communications log. The latter contains the detailed accounts of the information elicited in the interviews.

One group devised a form that was used at meetings of the group. The set of forms for all meetings was their log. A sample from this log is the next page.

Members : Cheryl Mester, David Miller, Sandra Mueller and Terry Ohm

Date : March 7 Time : 3:30

Who attended : Dave, Sandy Terry, Cheryl

Topics Discussed : Changing rooms and using waiting list - need of open and close status and how it works.
Controversy over specification of showing classes with description

Work completed & by Whom :

Class information Specification by Terry Ohm

Waiting list specification by Cheryl Mester

Add Drop and Registering by David Miller

Withdraw and Mail by Sandra Mueller

Work assigned & to whom :

Revision of waiting list to Cheryl Mester

Revision of Class information to Terry Ohm

Revision of Add Drop and Revision to David Miller

Revision of withdraw to Sandra Mueller

Other notes :

EXERCISES AND PROJECTS

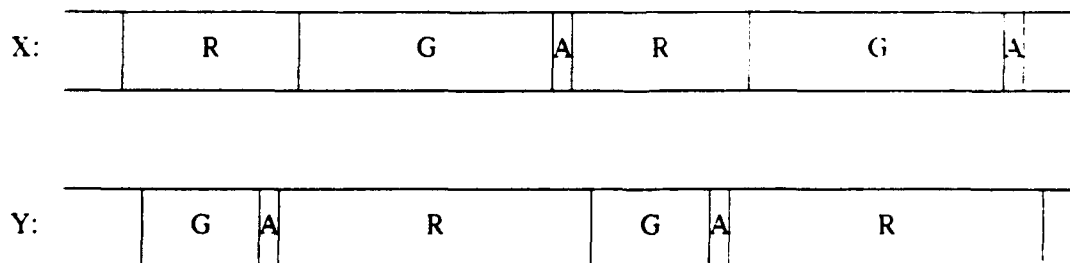
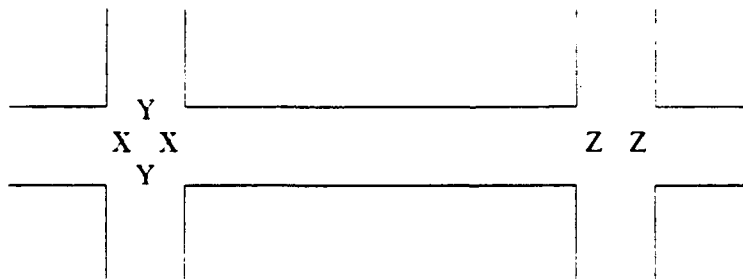
Small SF specifications. The first exercise can be the grade record for a class. Next is an appointments calendar with reminders. From this one can advance to an automobile maintenance data base that keeps track of all maintenance activities and reminds the driver of periodic checks. At about the same level of difficulty is a subscription system that accepts subscriptions for a journal, determines when renewal notices and follow-up notices are to be sent out, and takes care of explicit and implicit cancellations.

Extension of the elevator specification. The elevator specification given as an example of the use of SF can be refined. Thus, the case can be examined in which the agenda of an elevator is empty, but *nullweight* returns false. This sensor can actually be replaced by a real-valued sensor that indicates the weight of the contents of the elevator. Overloading of the elevator can then also be dealt with. A further extension should be provided to take care of recovery of the elevator from a power failure. Extend the given specification, but beware of "goldplating", namely a situation in which the system becomes too elaborate. For example, a person who has pressed a wrong button could wish for a means of cancelling this action. However, pressing all buttons is recommended as a countermeasure to being attacked on an elevator, and a cancelling feature would reduce the effectiveness of this measure.

Formal-wear-hire and dry-cleaning establishment. An establishment consists of a formal-wear-hire section and a laundry and dry-cleaning section. The formal-wear-hire section hires out bridal gowns, ball gowns, tuxedos, and similar garments. The hiring charge for each item is composed of a fixed charge and a variable charge determined by the length of the period of hire. The total variable charge for a hire transaction, which may involve several garments, depends also on the total fixed charge--the variable charge is reduced by a percentage proportional to the amount of the total fixed charge. All garments go to the laundry and dry-cleaning section on return, but this section also launders and dry-cleans for the general public. Develop an SF specification of this establishment. In

particular, determine how much (if anything) this specification has in common with the boat-hire specification, and devise an approach that takes advantage of any commonality, i.e., examine to what extent components of an SF specification can be made reusable.

Traffic lights. Traffic lights X go through a sequence ..., green, amber, red, green, The duration of the amber period is fixed, but the durations of the green and red periods are adjustable parameters. Lights Y are driven by lights X. The duration of the amber period is the same as for lights X, and there is a fixed overlap period at which both sets of lights show red. Write an SF specification for this system. Suppose lights Z are now installed. They are to be driven by lights X and are to be synchronized with them so that the green, amber, and red periods have the same durations for both lights, but the sequence for Z lags behind the sequence for X by some adjustable time interval. Does this modification need to cause any changes to the segment that controls lights X?



Automobile cruise control. This system is to maintain an automobile at a fixed cruising speed. A mechanism monitors the current speed of the car and adjusts the throttle setting

whenever the current speed has deviated too far from the selected cruising speed. There are three switches. The cruise control on/off switch engages or disengages the system. Cruise control may only be engaged when the engine is on, and it automatically disengages when the engine goes off. When the system is engaged, a cruising speed is selected by bringing the car to this speed and pressing a "select" button. Application of brakes cuts out cruise control, and the speed has to be controlled manually. Now, if the "select" button is pressed, the actual current speed becomes the new cruise speed, but if a "resume" button is pressed, then the system reverts to the cruising speed in effect before braking. Acceleration also overrides cruise control, but in this case resumption of the cruising speed is automatic. However, "select" can again be used to select a new cruising speed. Write a formal specification for this system.

Coin-operated luggage lockers. The basic usage of the lockers is as follows: (1) luggage is deposited in an unoccupied locker, and the door of the locker is closed; (2) an appropriate payment is made, the door is locked, and the user gets a key; (3) the key unlocks the locker door at any time within the next 24 hours, say, and the locker then reverts to unoccupied status. Many variants of this basic pattern exist. For example, in some French railway stations lockers come in sets of six, each set being provided with just one control mechanism. After the luggage has been deposited in an unoccupied locker and the door closed, turning of the door handle causes the mechanism to display the amount of money to be deposited. As coins are dropped in, the display shows how much remains to be paid. Hire is for 48 hours, and there is a return of change in case of overpayment. On receipt of payment the mechanism locks the door, prints the identifier of the locker and a numerical lock code on a slip of paper, and outputs this slip of paper. The door is unlocked by input of the lock code from a small keyboard that is part of the control mechanism. Note that during the time between the turning of the door handle and receipt of the lock code it should not be possible to turn the other five door handles. Hence it is essential that a transaction be terminated whenever this time interval exceeds a limit. Write a specification for this system. Write it in such a

way that the system can be easily converted to variable hiring times. Under this variant, as coins are dropped in, the display shows the length of time that the payment has bought this far.

Student registration. An SF specification of a student registration system is an open-ended project. It can be made as modest or as elaborate as time permits. The students should themselves decide what to include in their systems.

Merge of sorted lists. Write a specification for a data transformer that generates a sorted list from an input of two sorted lists.

The n-queens problem. Consider the problem of placing n queens on an $n \times n$ chess board so that they do not attack each other. The solution is a Boolean function Q .

$$Q: 1..n \times 1..n \rightarrow \text{Boolean}.$$

such that $Q(i, j)$ is true when square (i, j) holds a queen, and $Q(i, j)$ is false when square (i, j) does not hold a queen. The specification is to be a predicate $N\text{-queens}(n)$ that is true for every Q that solves the problem and false otherwise. (The implementation consists of finding actual functions Q that make $N\text{-queens}$ true.)

Solution:

$$\begin{aligned} N\text{-queens}(n) = & \\ & \text{Allop}(\wedge; \{ \\ & \quad \text{Allop}(\vee; \{Q(i, j) \\ & \quad \quad \wedge \text{Allop}(\wedge; \{\text{not}(Q(s, j)) \mid 1 \leq s \leq i-1 \vee i+1 \leq s \leq n\}) \\ & \quad \quad \wedge \text{Allop}(\wedge; \{\text{not}(Q(i, t)) \mid 1 \leq t \leq j-1 \vee j+1 \leq t \leq n\}) \\ & \quad \quad \wedge \text{Allop}(\wedge; \{s+t=i+j \vee s-t=i-j \rightarrow \text{not}(Q(s, t)) \\ & \quad \quad \quad \mid (1 \leq s \leq i-1 \vee i+1 \leq s \leq n) \wedge (1 \leq t \leq j-1 \vee j+1 \leq t \leq n)\}) \\ & \quad \mid 1 \leq j \leq n\}) \\ & \mid 1 \leq i \leq n\}); \end{aligned}$$

A spelling checker. Using the discussion of a spelling checker given on p.54 as a guide, develop predicative specifications of the components of the spelling checker.

Additional specification problems. Three interesting system descriptions that can be converted into formal specifications are given in the literature. They are a package routing system [Swartout and Balzer, *Comm CACM* 25 (1982), 438-440], a telephone dialing system [Dasarathy, *IEEE Trans. Software Eng.* SE-11 (1985), 80-86], and a furnace controller [From the problem set in *Proc. 4th International Workshop on Software Specification and Design, 1987* (Harandi, ed.)].

UNLIMITED, UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-SM-8-1.0			7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE		
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.		6b. OFFICE SYMBOL (If applicable) SEI		7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE, MA 01731	
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE		8b. OFFICE SYMBOL (If applicable) ESD/ AVS	
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003			
11. TITLE (Include Security Classification) Formal Specification of Software		10. SOURCE OF FUNDING NOS.			
PERSONAL AUTHOR(S) Alfs Berztiss, University of Pittsburgh		10. SOURCE OF FUNDING NOS.			
13a. TYPE OF REPORT FINAL		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) October 1987	
15. PAGE COUNT 75		16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)			
FIELD	GROUP	SUB GR	formal specification verification		
			software specification formal method		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
These materials support the SEI curriculum module SEI-CM-8 "Formal Specification of Software."					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION		
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF			22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630		22c. OFFICE SYMBOL ESD/AVS (SEI JPO)

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials package* (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials package* (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Permission to make copies or derivative works of SEI curriculum modules, support materials, and educational materials is granted, without fee, provided that the copies and derivative works are not made or distributed for direct commercial advantage, and that all copies and derivative works cite the original document by name, author's name, and document number and give notice that the copying is by permission of Carnegie Mellon University.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

- CM-1 [superseded by CM-19]
- CM-2 Introduction to Software Design
- CM-3 The Software Technical Review Process*
- CM-4 Software Configuration Management*
- CM-5 Information Protection
- CM-6 Software Safety
- CM-7 Assurance of Software Quality
- CM-8 Formal Specification of Software*
- CM-9 Unit Testing and Analysis
- CM-10 Models of Software Evolution: Life Cycle and Process
- CM-11 Software Specifications: A Framework
- CM-12 Software Metrics
- CM-13 Introduction to Software Verification and Validation
- CM-14 Intellectual Property Protection for Software
- CM-15 Software Development and Licensing Contracts
- CM-16 Software Development Using VDM
- CM-17 User Interface Development*
- CM-18 [superseded by CM-23]
- CM-19 Software Requirements
- CM-20 Formal Verification of Programs
- CM-21 Software Project Management
- CM-22 Software Design Methods for Real-Time Systems*
- CM-23 Technical Writing for Software Engineers
- CM-24 Concepts of Concurrent Programming
- CM-25 Language and System Support for Concurrent Programming*
- CM-26 Understanding Program Dependencies

Educational Materials

- EM-1 Software Maintenance Exercises for a Software Engineering Project Course
- EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education
- EM-3 Reading Computer Programs: Instructor's Guide and Exercises